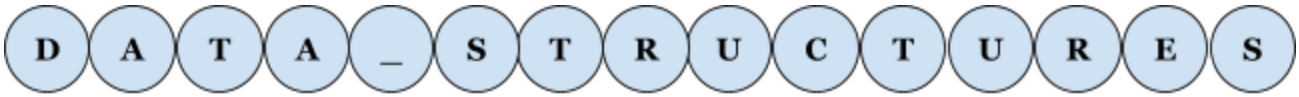
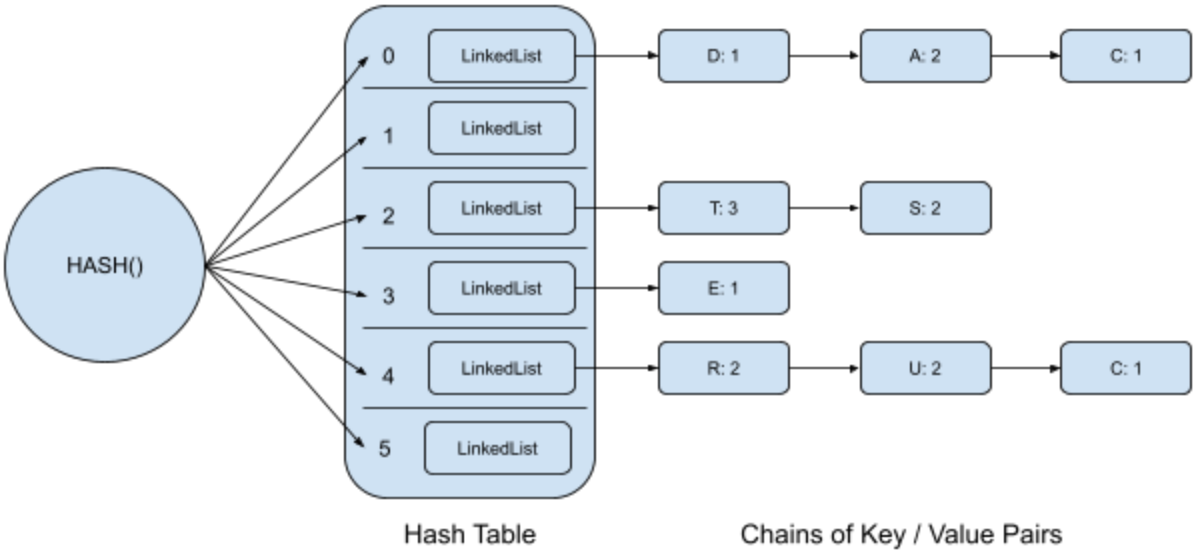
CS261 Data Structures

Assignment 5

v 1.03 (revised 2/23/2021)

Your Very Own HashMap, MinHeap, and AVL Tree





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 default parameters to method definitions.

   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 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.

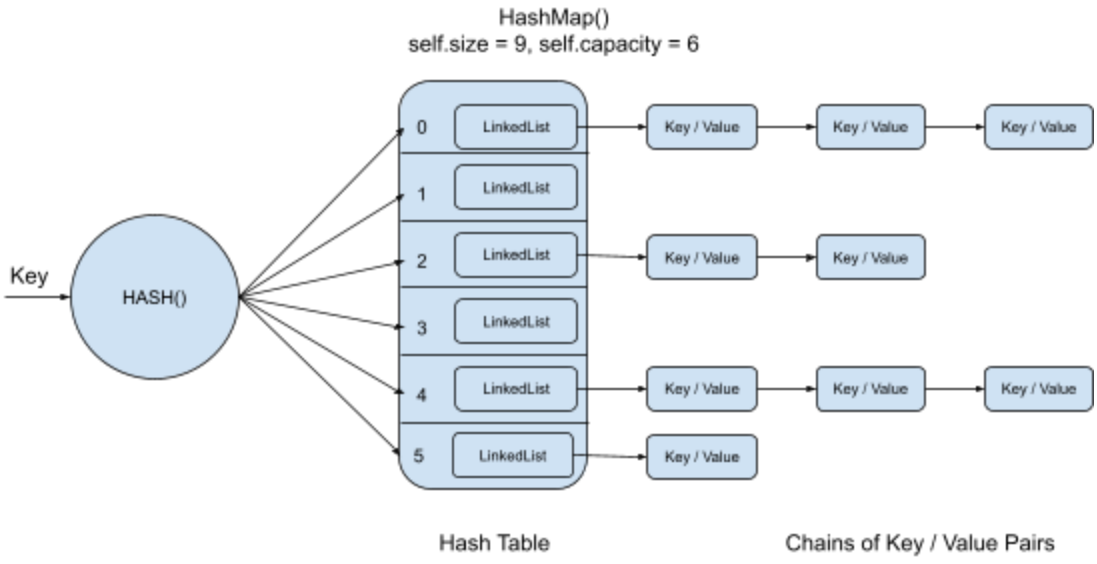
8. We will test your implementation with different types of objects, not just integers. We guarantee that all such objects will have correct implementation of methods __eq__, __lt__, __gt__, __ge__, __le__ and __str__.

Part 1 - Summary and Specific Instructions

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

```
put()
get()
remove()
contains_key()
clear()
empty_buckets()
resize_table()
table_load()
get_keys()
```

2. Use a dynamic array to store your hash table. Implement chaining for collision resolution using a singly linked list. Chains of key / value pairs must be stored in linked list nodes. Diagram below illustrates the overall architecture of the HashMap.



3. Two pre-written classes are provided for you in the skeleton code - DynamicArray and LinkedList (file `a5_include.py`). You should use objects of these classes in your HashMap class implementation. Use a DynamicArray object to store your hash table and LinkedList objects to store chains of key / value pairs.

4. Provided DynamicArray and LinkedList classes may provide different functionality than those described in the lectures or implemented in prior homework assignments. Review docstrings in the skeleton code to understand available methods, their use, and input / output parameters.

5. The number of objects stored in the HashMap will be between 0 and 1,000,000 inclusive.

6. Two pre-written hash functions are provided in the skeleton code. Make sure you test your code with both functions. We will use these two functions in our testing of your implementation.

7. RESTRICTIONS: You are NOT allowed to use ANY built-in Python data structures and/or their methods.

   You are NOT allowed to directly access any variables of the DynamicArray or LinkedList classes. All work must be done only by using class methods.

8. Variables in the HashMap and SLNode 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 the HashMap or SLNode classes.

empty_buckets(self) -> int:

This method returns a number of empty buckets in the hash table.

Example #1:

```
m = HashMap(100, hash_function_1)
print(m.empty_buckets(), m.size, m.capacity)
m.put('key1', 10)
print(m.empty_buckets(), m.size, m.capacity)
m.put('key2', 20)
print(m.empty_buckets(), m.size, m.capacity)
m.put('key1', 30)
print(m.empty_buckets(), m.size, m.capacity)
m.put('key4', 40)
print(m.empty_buckets(), m.size, m.capacity)
```

Output:

```
100 0 100
99 1 100
98 2 100
98 2 100
97 3 100
```

Example #2:

```
# this test assumes that put() has already been correctly implemented
m = HashMap(50, hash_function_1)
    for i in range(150):
        m.put('key' + str(i), i * 100)
        if i % 30 == 0:
            print(m.empty_buckets(), m.size, m.capacity)
```

Output:

```
49 1 50
39 31 50
36 61 50
33 91 50
30 121 50
```

table_load(self) -> float:

This method returns the current hash table load factor.

Example #1:

```
m = HashMap(100, hash_function_1)
print(m.table_load())
m.put('key1', 10)
print(m.table_load())
m.put('key2', 20)
print(m.table_load())
m.put('key1', 30)
print(m.table_load())
```

Output:

```
0.0
0.01
0.02
0.02
```

Example #2:

```
m = HashMap(50, hash_function_1)
    for i in range(50):
        m.put('key' + str(i), i * 100)
        if i % 10 == 0:
            print(m.table_load(), m.size, m.capacity)
```

Output:

```
0.02 1 50
0.22 11 50
0.42 21 50
0.62 31 50
0.82 41 50
```

clear(self) -> None:

This method clears the content of the hash map. It does not change underlying hash table capacity.

Example #1:

```
m = HashMap(100, hash_function_1)
print(m.size, m.capacity)
m.put('key1', 10)
m.put('key2', 20)
m.put('key1', 30)
print(m.size, m.capacity)
m.clear()
print(m.size, m.capacity)
```

Output:

```
0 100
2 100
0 100
```

Example #2:

```
m = HashMap(50, hash_function_1)
print(m.size, m.capacity)
m.put('key1', 10)
print(m.size, m.capacity)
m.put('key2', 20)
print(m.size, m.capacity)
m.resize_table(100)
print(m.size, m.capacity)
m.clear()
print(m.size, m.capacity)
```

Output:

```
0 50
1 50
2 50
2 100
0 100
```

put(self, key: str, value: object) -> None:

This method updates the key / value pair in the hash map. If a given key already exists in the hash map, its associated value should be replaced with the new value. If a given key is not in the hash map, a key / value pair should be added.

Example #1:

```
m = HashMap(50, hash_function_1)
for i in range(150):
    m.put('str' + str(i), i * 100)
    if i % 25 == 24:
        print(m.empty_buckets(), m.table_load(), m.size, m.capacity)
```

Output:

```
39 0.5 25 50
37 1.0 50 50
35 1.5 75 50
32 2.0 100 50
30 2.5 125 50
30 3.0 150 50
```

Example #2:

```
m = HashMap(40, hash_function_2)
for i in range(50):
    m.put('str' + str(i // 3), i * 100)
    if i % 10 == 9:
        print(m.empty_buckets(), m.table_load(), m.size, m.capacity)
```

Output:

```
36 0.1 4 40
33 0.175 7 40
30 0.25 10 40
27 0.35 14 40
25 0.425 17 40
```

contains_key(self, key: str) -> bool:

This method returns True if the given key is in the hash map, otherwise it returns False. An empty hash map does not contain any keys.

Example #1:

```
m = HashMap(50, hash_function_1)
print(m.contains_key('key1'))
m.put('key1', 10)
m.put('key2', 20)
m.put('key3', 30)
print(m.contains_key('key1'))
print(m.contains_key('key4'))
print(m.contains_key('key2'))
print(m.contains_key('key3'))
m.remove('key3')
print(m.contains_key('key3'))
```

Output:

```
False
True
False
True
True
False
```

Example #2:

```
m = HashMap(75, hash_function_2)
keys = [i for i in range(1, 1000, 20)]
for key in keys:
    m.put(str(key), key * 42)
print(m.size, m.capacity)
result = True
for key in keys:
    # all inserted keys must be present
    result &= m.contains_key(str(key))
    # NOT inserted keys must be absent
    result &= not m.contains_key(str(key + 1))
print(result)
```

Output:

```
50 75
True
```

get(self, key: str) -> object:

This method returns the value associated with the given key. If the key is not in the hash map, the method returns None.

Example #1:

```
m = HashMap(30, hash_function_1)
print(m.get('key'))
m.put('key1', 10)
print(m.get('key1'))
```

Output:

```
None
10
```

Example #2:

```
m = HashMap(150, hash_function_2)
for i in range(200, 300, 7):
    m.put(str(i), i * 10)
print(m.size, m.capacity)
for i in range(200, 300, 21):
    print(i, m.get(str(i)), m.get(str(i)) == i * 10)
    print(i + 1, m.get(str(i + 1)), m.get(str(i + 1)) == (i + 1) * 10)
```

Output:

```
15 150
200 2000 True
201 None False
221 2210 True
222 None False
242 2420 True
243 None False
263 2630 True
264 None False
284 2840 True
285 None False
```

remove(self, key: str) -> None:

This method removes the given key and its associated value from the hash map. If a given key is not in the hash map, the method does nothing (no exception needs to be raised).

Example #1:

```
m = HashMap(50, hash_function_1)
print(m.get('key1'))
m.put('key1', 10)
print(m.get('key1'))
m.remove('key1')
print(m.get('key1'))
m.remove('key4')
```

Output:

```
None
10
None
```

resize_table(self, new_capacity: int) -> None:

This method changes the capacity of the internal hash table. All existing key / value pairs must remain in the new hash map and all hash table links must be rehashed. If new_capacity is less than 1, this method should do nothing.

Example #1:

```
m = HashMap(20, hash_function_1)
m.put('key1', 10)
print(m.size, m.capacity, m.get('key1'), m.contains_key('key1'))
m.resize_table(30)
print(m.size, m.capacity, m.get('key1'), m.contains_key('key1'))
```

Output:

```
1 20 10 True
1 30 10 True
```

Example #2:

```
m = HashMap(75, hash_function_2)
keys = [i for i in range(1, 1000, 13)]
for key in keys:
    m.put(str(key), key * 42)
print(m.size, m.capacity)

for capacity in range(111, 1000, 117):
    m.resize_table(capacity)
    m.put('some key', 'some value')
    result = m.contains_key('some key')
    m.remove('some key')
    for key in keys:
        result &= m.contains_key(str(key))
        result &= not m.contains_key(str(key + 1))
    print(capacity, result, m.size, m.capacity, round(m.table_load(), 2))
```

Output:

```
77 75
111 True 77 111 0.69
228 True 77 228 0.34
345 True 77 345 0.22
462 True 77 462 0.17
579 True 77 579 0.13
696 True 77 696 0.11
813 True 77 813 0.09
930 True 77 930 0.08
```

get_keys(self) -> DynamicArray:

This method returns a DynamicArray that contains all keys stored in your hash map. The order of the keys in the DA does not matter.

Example #1:

```
m = HashMap(10, hash_function_2)
for i in range(100, 200, 10):
    m.put(str(i), str(i * 10))
print(m.get_keys())

m.resize_table(1)
print(m.get_keys())

m.put('200', '2000')
m.remove('100')
m.resize_table(2)
print(m.get_keys())
```

Output:

```
['160', '110', '170', '120', '180', '130', '190', '140', '150', '100']
['100', '150', '140', '190', '130', '180', '120', '170', '110', '160']
['200', '160', '110', '170', '120', '180', '130', '190', '140', '150']
```

Part 2 - Summary and Specific Instructions

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

   ```
   add()
   get_min()
   remove_min()
   build_heap()
   ```

2. Prewritten DynamicArray class is provided for you in the skeleton code (file `a5_include.py`). You should use objects of this class in your MinHeap class implementation for storing heap content.

3. Prewritten DynamicArray class may provide different functionality than that described in the lectures or implemented in prior homework assignments. Review docstrings in the skeleton code to understand available methods, their use, and input / output parameters.

4. The number of objects stored in the MinHeap will be between 0 and 1,000,000 inclusive.

5. RESTRICTIONS: You are NOT allowed to use ANY built-in Python data structures and/or their methods.

   You are NOT allowed to directly access any variables of the DynamicArray class. All work must be done only by using class methods.

6. Variables in the MinHeap 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 MinHeap class.

7. You may assume all methods of the provided DynamicArray class have O(1) runtime complexity.

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

This method adds a new object to the MinHeap maintaining heap property.

Runtime complexity of this implementation must be O(logN).

Example #1:

```
h = MinHeap()
print(h, h.is_empty())
for value in range(300, 200, -15):
    h.add(value)
    print(h)
```

Output:

```
HEAP [] True
HEAP [300]
HEAP [285, 300]
HEAP [270, 300, 285]
HEAP [255, 270, 285, 300]
HEAP [240, 255, 285, 300, 270]
HEAP [225, 255, 240, 300, 270, 285]
HEAP [210, 255, 225, 300, 270, 285, 240]
```

Example #2:

```
h = MinHeap(['fish', 'bird'])
print(h)
for value in ['monkey', 'zebra', 'elephant', 'horse', 'bear']:
    h.add(value)
    print(h)
```

Output:

```
HEAP ['bird', 'fish']
HEAP ['bird', 'fish', 'monkey']
HEAP ['bird', 'fish', 'monkey', 'zebra']
HEAP ['bird', 'elephant', 'monkey', 'zebra', 'fish']
HEAP ['bird', 'elephant', 'horse', 'zebra', 'fish', 'monkey']
HEAP ['bear', 'elephant', 'bird', 'zebra', 'fish', 'monkey', 'horse']
```

get_min(self) -> object:

This method returns an object with a minimum key without removing it from the heap. If the heap is empty, the method raises a MinHeapException.

Runtime complexity of this implementation must be O(1).

Example #1:

```
h = MinHeap(['fish', 'bird'])
print(h)
print(h.get_min(), h.get_min())
```

Output:

```
HEAP ['bird', 'fish']
bird bird
```

remove_min(self) -> object:

This method returns an object with a minimum key and removes it from the heap. If the heap is empty, the method raises a MinHeapException.

When doing a downward percolation of the replacement node, if both children of the node have the same values (both smaller than the node), swap with the left child.

Runtime complexity of this implementation must be O(logN).

Example #1:

```
h = MinHeap([1, 10, 2, 9, 3, 8, 4, 7, 5, 6])
while not h.is_empty():
    print(h, end=' ')
    print(h.remove_min())
```

Output:

```
HEAP [1, 3, 2, 5, 6, 8, 4, 10, 7, 9] 1
HEAP [2, 3, 4, 5, 6, 8, 9, 10, 7] 2
HEAP [3, 5, 4, 7, 6, 8, 9, 10] 3
HEAP [4, 5, 8, 7, 6, 10, 9] 4
HEAP [5, 6, 8, 7, 9, 10] 5
HEAP [6, 7, 8, 10, 9] 6
HEAP [7, 9, 8, 10] 7
HEAP [8, 9, 10] 8
HEAP [9, 10] 9
HEAP [10] 10
```

build_heap(self, da: DynamicArray) -> None:

This method receives a dynamic array with objects in any order and builds a proper MinHeap from them. Current content of the MinHeap is lost.

Runtime complexity of this implementation must be O(N). If the runtime complexity is O(NlogN), you will not receive any points for this portion of the assignment, even if you pass Gradescope.

Example #1:

```
da = DynamicArray([100, 20, 6, 200, 90, 150, 300])
h = MinHeap(['zebra', 'apple'])
print(h)
h.build_heap(da)
print(h)
da.set_at_index(0, 500)
print(da)
print(h)
```

Output:

```
HEAP ['apple', 'zebra']
HEAP [6, 20, 100, 200, 90, 150, 300]
[500, 20, 6, 200, 90, 150, 300]
HEAP [6, 20, 100, 200, 90, 150, 300]
```

Part 3 - Summary and Specific Instructions

1. Implement the AVL class by completing provided skeleton code in the file `avl.py`. In this part of the assignment you will not write an entire AVL class from scratch. AVL class will extend the BST class that you have written in the prior assignment and just overwrite (overload) two main methods: `add()` and `remove()`.

   Remember, that AVL tree is really just a BST tree with some additional code that keeps it balanced during add and remove operations. So once completed, your implementation will include all the same methods as the previously written BST class, but the tree will stay balanced at all times.

2. When reviewing provided skeleton code, please note that TreeNode class has a couple of additional 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 100,000 inclusive.

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, BST 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.

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

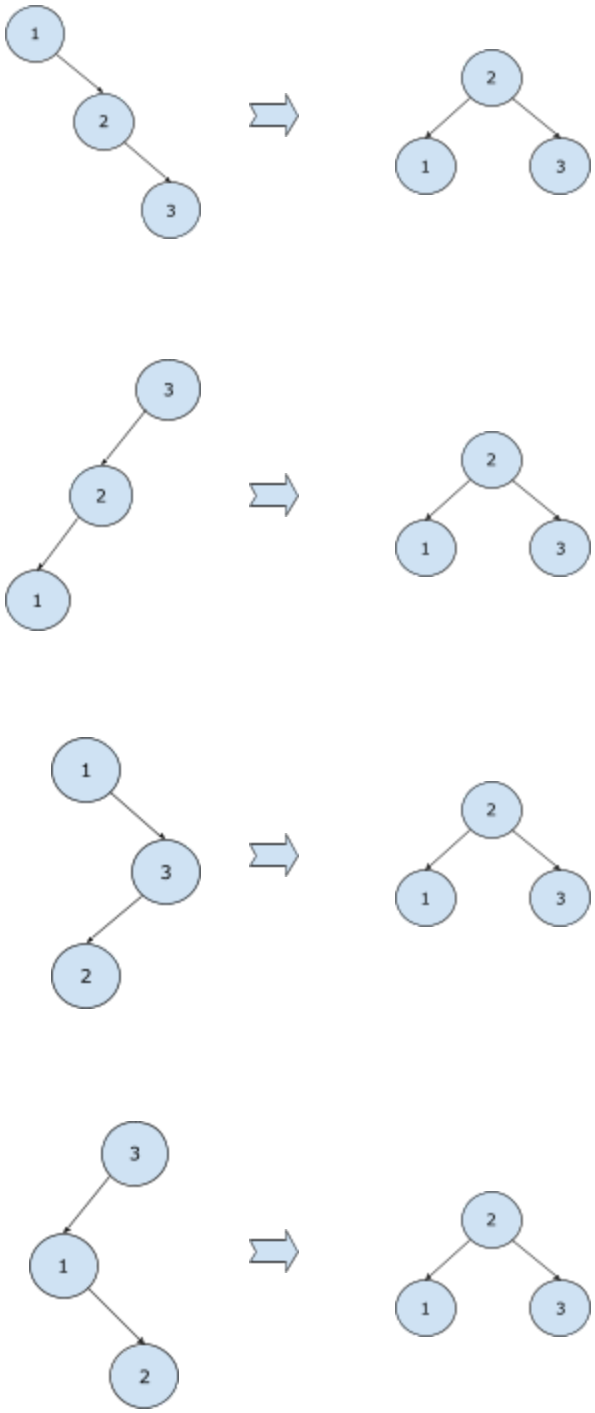
This method adds new value to the tree, maintaining AVL property. Duplicates must not be allowed. If the value is already in the tree, the method should do nothing.

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:

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



Example #2:

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

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

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

NOTE: See 'Specific Instructions' for 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  : TREE pre-order { 2, 1, 3 } DEL: 1
RESULT : TREE pre-order { 2, 3 }
INPUT  : TREE pre-order { 2, 1, 3 } DEL: 2
RESULT : TREE pre-order { 3, 1 }
INPUT  : TREE pre-order { 2, 1, 3 } DEL: 3
RESULT : TREE pre-order { 2, 1 }
INPUT  : TREE pre-order { 50, 30, 20, 40, 45, 70, 60, 80 } DEL: 0
RESULT : TREE pre-order { 50, 30, 20, 40, 45, 70, 60, 80 }
INPUT  : TREE pre-order { 50, 30, 20, 40, 45, 70, 60, 80 } DEL: 45
RESULT : TREE pre-order { 50, 30, 20, 40, 70, 60, 80 }
INPUT  : TREE pre-order { 50, 30, 20, 40, 45, 70, 60, 80 } DEL: 40
RESULT : TREE pre-order { 50, 30, 20, 45, 70, 60, 80 }
INPUT  : TREE pre-order { 50, 30, 20, 40, 45, 70, 60, 80 } DEL: 30
RESULT : TREE 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  : TREE pre-order { 50, 30, 20, 40, 45, 70, 60, 80 } DEL: 20
RESULT : TREE pre-order { 50, 40, 30, 45, 70, 60, 80 }
INPUT  : TREE pre-order { 50, 30, 20, 15, 40, 70, 60, 80 } DEL: 40
RESULT : TREE pre-order { 50, 20, 15, 30, 70, 60, 80 }
INPUT  : TREE pre-order { 50, 30, 20, 40, 35, 70, 60, 80 } DEL: 20
RESULT : TREE pre-order { 50, 35, 30, 40, 70, 60, 80 }
INPUT  : TREE pre-order { 50, 30, 20, 25, 40, 70, 60, 80 } DEL: 40
RESULT : TREE 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  : TREE pre-order { 5, -3, -7, -9, -5, 1, -1, 3, 9, 7, 13, 11, 15 } -9
RESULT : TREE pre-order { 5, -3, -7, -5, 1, -1, 3, 9, 7, 13, 11, 15 }
INPUT  : TREE pre-order { 5, -3, -7, -5, 1, -1, 3, 9, 7, 13, 11, 15 } -7
RESULT : TREE pre-order { 5, -3, -5, 1, -1, 3, 9, 7, 13, 11, 15 }
INPUT  : TREE pre-order { 5, -3, -5, 1, -1, 3, 9, 7, 13, 11, 15 } -5
RESULT : TREE pre-order { 5, 1, -3, -1, 3, 9, 7, 13, 11, 15 }
INPUT  : TREE pre-order { 5, 1, -3, -1, 3, 9, 7, 13, 11, 15 } -3
RESULT : TREE pre-order { 5, 1, -1, 3, 9, 7, 13, 11, 15 }
INPUT  : TREE pre-order { 5, 1, -1, 3, 9, 7, 13, 11, 15 } -1
RESULT : TREE pre-order { 5, 1, 3, 9, 7, 13, 11, 15 }
INPUT  : TREE pre-order { 5, 1, 3, 9, 7, 13, 11, 15 } 1
RESULT : TREE pre-order { 9, 5, 3, 7, 13, 11, 15 }
INPUT  : TREE pre-order { 9, 5, 3, 7, 13, 11, 15 } 3
RESULT : TREE pre-order { 9, 5, 7, 13, 11, 15 }
INPUT  : TREE pre-order { 9, 5, 7, 13, 11, 15 } 5
RESULT : TREE pre-order { 9, 7, 13, 11, 15 }
INPUT  : TREE pre-order { 9, 7, 13, 11, 15 } 7
RESULT : TREE pre-order { 13, 9, 11, 15 }
INPUT  : TREE pre-order { 13, 9, 11, 15 } 9
RESULT : TREE pre-order { 13, 11, 15 }
INPUT  : TREE pre-order { 13, 11, 15 } 11
RESULT : TREE pre-order { 13, 15 }
INPUT  : TREE pre-order { 13, 15 } 13
RESULT : TREE pre-order { 15 }
INPUT  : TREE pre-order { 15 } 15
RESULT : TREE 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  : 12 TREE pre-order { 21, 9, 3, 0, 6, 15, 12, 18, 27, 24, 30, 33 } 21
RESULT : TREE pre-order { 24, 9, 3, 0, 6, 15, 12, 18, 30, 27, 33 }
INPUT  : 11 TREE pre-order { 24, 9, 3, 0, 6, 15, 12, 18, 30, 27, 33 } 24
RESULT : TREE pre-order { 27, 9, 3, 0, 6, 15, 12, 18, 30, 33 }
INPUT  : 10 TREE pre-order { 27, 9, 3, 0, 6, 15, 12, 18, 30, 33 } 27
RESULT : TREE pre-order { 9, 3, 0, 6, 30, 15, 12, 18, 33 }
INPUT  : 9 TREE pre-order { 9, 3, 0, 6, 30, 15, 12, 18, 33 } 9
RESULT : TREE pre-order { 12, 3, 0, 6, 30, 15, 18, 33 }
INPUT  : 8 TREE pre-order { 12, 3, 0, 6, 30, 15, 18, 33 } 12
RESULT : TREE pre-order { 15, 3, 0, 6, 30, 18, 33 }
INPUT  : 7 TREE pre-order { 15, 3, 0, 6, 30, 18, 33 } 15
RESULT : TREE pre-order { 18, 3, 0, 6, 30, 33 }
INPUT  : 6 TREE pre-order { 18, 3, 0, 6, 30, 33 } 18
RESULT : TREE pre-order { 30, 3, 0, 6, 33 }
INPUT  : 5 TREE pre-order { 30, 3, 0, 6, 33 } 30
RESULT : TREE pre-order { 3, 0, 33, 6 }
INPUT  : 4 TREE pre-order { 3, 0, 33, 6 } 3
RESULT : TREE pre-order { 6, 0, 33 }
INPUT  : 3 TREE pre-order { 6, 0, 33 } 6
RESULT : TREE pre-order { 33, 0 }
```

Example #5:

```
for _ in range(100):
    case = list(set(random.randrange(1, 20000) for _ in range(900)))
    avl = AVL(case)
    if avl.size() != len(case):
        raise Exception("PROBLEM WITH ADD OPERATION")
    for value in case[::2]:
        avl.remove(value)
    if avl.size() != len(case) - len(case[::2]):
        raise Exception("PROBLEM WITH REMOVE OPERATION")
print('Stress test finished')
```

Output:

```
Stress test finished
```

Comprehensive Example #1:

```
tree = AVL()
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       1        2        3   True       True     True
   15      4       2        2        4   False      False    False
   12      5       2        3        5   False      True     False
    8      6       2        3        6   False      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 }
```

Comprehensive Example #2:

```
tree = AVL()
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          3       1        2        3   True       True     True
           4       2        2        4   True       False    False
S          5       2        2        5   False      False    False
T          5       2        2        5   False      False    False
R          6       2        3        6   False      False    False
U          7       3        3        7   False      False    False
C          8       3        4        8   False      False    False
T          8       3        4        8   False      False    False
U          8       3        4        8   False      False    False
R          8       3        4        8   False      False    False
E          9       3        4        9   False      False    False
S          9       3        4        9   False      False    False

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