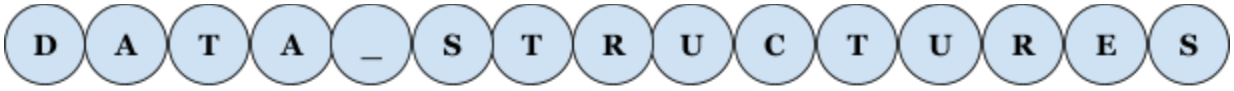
CS261 Data Structures

Assignment 2

Spring 2023

Dynamic Array Implementation
(plus a brand new Bag)



---

da = DynamicArray(list("DATA"))

self.size = 4
self.capacity = 4
self.data = ['D', 'A', 'T', 'A']

da = DynamicArray(list("STRUCTURES"))

self.size = 10
self.capacity = 16
self.data = ['S', 'T', 'R', 'U', 'C', 'T', 'U', 'R', 'E', 'S', None, None, None, None, None, None]

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. Gradescope allows you to choose which submission will be graded.

2. In Gradescope, your code will be 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 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, write comments throughout your 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 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 methods and variables, as needed.

   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.

   Half points will be deducted from DynamicArray's `find_mode()` for the incorrect time complexity. All points will be deducted from Bag methods for the incorrect time complexity.

   Note that the `__iter__()` method in the Bag implementation **will** require you to add variables in that method, and this is permitted.

6. 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 a 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 are required to use an iterative solution.** Recursion is not permitted. We will specify the maximum input size that your solution must handle.

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

Part 1 - Summary and Specific Instructions

1. Implement a DynamicArray class by completing the skeleton code provided in the file `dynamic_array.py`. The DynamicArray class will use a StaticArray object as its underlying data storage container, and will provide many methods similar to the functionality of Python lists. Once completed, your implementation will include the following methods:

   ```
   resize()
   append()
   insert_at_index()
   remove_at_index()
   slice()
   merge()
   map()
   filter()
   reduce()
   ```

   * Several class methods, like `is_empty()`, `length()`, `get_at_index()`, and `set_at_index()` have been pre-written for you.

   * The `dynamic_array.py` file also contains the `find_mode()` function, but this is a separate function outside the class that you will need to implement.

2. We will test your implementations 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 array at any given time will be between 0 and 1,000,000 inclusive. An array must allow for the storage of duplicate objects.

4. Variables in the DynamicArray class are marked as private, so they may only be accessed and/or changed directly inside the class. Use the provided getter or setter methods when you need this functionality outside of the DynamicArray class.

5. RESTRICTIONS: You are NOT allowed to use ANY built-in Python data structures and/or their methods in any of your solutions. This includes built-in Python lists, dictionaries, or anything else. You must solve this portion of the assignment by importing and using objects of the StaticArray class (prewritten for you), and using class methods to write your solution.

   You are also not allowed to directly access any variables of the StaticArray class (e.g. `self._size` or `self._data`). Access to StaticArray variables must only be done by using the StaticArray class methods. **Don't forget to include your StaticArray class from Assignment 1 in your project.**

   Read the *Coding Guides and Tips* module for a detailed description of these topics.

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

This method changes the capacity of the underlying storage for the elements in the dynamic array. It does not change the values or the order of any elements currently stored in the array.

It is intended to be an **"internal"** method of the DynamicArray class, called by other class methods such as `append()`, `remove_at_index()`, or `insert_at_index()`, to manage the capacity of the underlying data structure.

The method should only accept positive integers for `new_capacity`. Additionally, `new_capacity` cannot be smaller than the number of elements currently stored in the dynamic array (which is tracked by the `self._size` variable). If `new_capacity` is not a positive integer, or if `new_capacity` is less than `self._size`, this method should not do any work and immediately exit.

Example #1:

```
da = DynamicArray()
da.print_da_variables()
da.resize(8)
da.print_da_variables()
da.resize(2)
da.print_da_variables()
da.resize(0)
da.print_da_variables()
```

Output:

```
Length: 0, Capacity: 4, STAT_ARR Size: 4 [None, None, None, None]
Length: 0, Capacity: 8, STAT_ARR Size: 8 [None, None, None, None, None, None,
None, None]
Length: 0, Capacity: 2, STAT_ARR Size: 2 [None, None]
Length: 0, Capacity: 2, STAT_ARR Size: 2 [None, None]
```

NOTE: Example 2 below will not work properly unless the `append()` method is implemented.

Example #2:

```
da = DynamicArray([1, 2, 3, 4, 5, 6, 7, 8])
print(da)
da.resize(20)
print(da)
da.resize(4)
print(da)
```

Output:

```
DYN_ARR Size/Cap: 8/8 [1, 2, 3, 4, 5, 6, 7, 8]
DYN_ARR Size/Cap: 8/20 [1, 2, 3, 4, 5, 6, 7, 8]
DYN_ARR Size/Cap: 8/20 [1, 2, 3, 4, 5, 6, 7, 8]
```

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

This method adds a new value at the end of the dynamic array. If the internal storage associated with the dynamic array is already full, you will need to DOUBLE its capacity before adding a new value (hint: you can use your already written `resize()` function for this).

Example #1:

```
da = DynamicArray()
da.print_da_variables()
da.append(1)
da.print_da_variables()
print(da)
```

Output:

```
Length: 0, Capacity: 4, STAT_ARR Size: 4 [None, None, None, None]
Length: 1, Capacity: 4, STAT_ARR Size: 4 [1, None, None, None]
DYN_ARR Size/Cap: 1/4 [1]
```

Example #2:

```
da = DynamicArray()
for i in range(9):
    da.append(i + 101)
    print(da)
```

Output:

```
DYN_ARR Size/Cap: 1/4 [101]
DYN_ARR Size/Cap: 2/4 [101, 102]
DYN_ARR Size/Cap: 3/4 [101, 102, 103]
DYN_ARR Size/Cap: 4/4 [101, 102, 103, 104]
DYN_ARR Size/Cap: 5/8 [101, 102, 103, 104, 105]
DYN_ARR Size/Cap: 6/8 [101, 102, 103, 104, 105, 106]
DYN_ARR Size/Cap: 7/8 [101, 102, 103, 104, 105, 106, 107]
DYN_ARR Size/Cap: 8/8 [101, 102, 103, 104, 105, 106, 107, 108]
DYN_ARR Size/Cap: 9/16 [101, 102, 103, 104, 105, 106, 107, 108, 109]
```

Example #3:

```
da = DynamicArray()
for i in range(600):
    da.append(i)
print(da.length())
print(da.get_capacity())
```

Output:

```
600
1024
```

insert_at_index(self, index: int, value: object) -> None:

This method adds a new value at the specified index in the dynamic array. Index 0 refers to the beginning of the array. If the provided index is invalid, the method raises a custom "DynamicArrayException". Code for the exception is provided in the skeleton file. If the array contains N elements, valid indices for this method are [0, N] inclusive.

If the internal storage associated with the dynamic array is already full, you will need to DOUBLE its capacity before adding a new value.

Example #1:

```
da = DynamicArray([100])
print(da)
da.insert_at_index(0, 200)
da.insert_at_index(0, 300)
da.insert_at_index(0, 400)
print(da)
da.insert_at_index(3, 500)
print(da)
da.insert_at_index(1, 600)
print(da)
```

Output:

```
DYN_ARR Size/Cap: 1/4 [100]
DYN_ARR Size/Cap: 4/4 [400, 300, 200, 100]
DYN_ARR Size/Cap: 5/8 [400, 300, 200, 500, 100]
DYN_ARR Size/Cap: 6/8 [400, 600, 300, 200, 500, 100]
```

Example #2:

```
da = DynamicArray()
try:
    da.insert_at_index(-1, 100)
except Exception as e:
    print("Exception raised:", type(e))
da.insert_at_index(0, 200)
try:
    da.insert_at_index(2, 300)
except Exception as e:
    print("Exception raised:", type(e))
print(da)
```

Output:

```
Exception raised: <class '__main__.DynamicArrayException'>
Exception raised: <class '__main__.DynamicArrayException'>
DYN_ARR Size/Cap: 1/4 [200]
```

Example #3:

```
da = DynamicArray()
for i in range(1, 10):
    index, value = i - 4, i * 10
    try:
        da.insert_at_index(index, value)
    except Exception as e:
        print("Cannot insert value", value, "at index", index)
print(da)
```

Output:

```
Cannot insert value 10 at index -3
Cannot insert value 20 at index -2
Cannot insert value 30 at index -1
DYN_ARR Size/Cap: 6/8 [40, 50, 60, 70, 80, 90]
```

remove_at_index(self, index: int) -> None:

This method removes the element at the specified index in the dynamic array. Index 0 refers to the beginning of the array. If the provided index is invalid, the method raises a custom "DynamicArrayException". Code for the exception is provided in the skeleton file. If the array contains N elements, valid indices for this method are [0, N - 1] inclusive.

When the number of elements stored in the array (before removal) is STRICTLY LESS than ¼ of its current capacity, the capacity must be reduced to TWICE the number of current elements. This check and capacity adjustment must occur BEFORE removal of the element.

If the current capacity (before reduction) is 10 elements or less, reduction should not occur at all. If the current capacity (before reduction) is greater than 10 elements, the reduced capacity cannot become less than 10 elements. Please see the examples below, especially example #3, for clarification.

Future assignments will depend on this method being able to run in O(1) best case.

Example #1:

```
da = DynamicArray([10, 20, 30, 40, 50, 60, 70, 80])
print(da)
da.remove_at_index(0)
print(da)
da.remove_at_index(6)
print(da)
da.remove_at_index(2)
print(da)
```

Output:

```
DYN_ARR Size/Cap: 8/8 [10, 20, 30, 40, 50, 60, 70, 80]
DYN_ARR Size/Cap: 7/8 [20, 30, 40, 50, 60, 70, 80]
DYN_ARR Size/Cap: 6/8 [20, 30, 40, 50, 60, 70]
DYN_ARR Size/Cap: 5/8 [20, 30, 50, 60, 70]
```

Example #2:

```
da = DynamicArray([1024])
print(da)
for i in range(17):
    da.insert_at_index(i, i)
print(da.length(), da.get_capacity())
for i in range(16, -1, -1):
    da.remove_at_index(0)
print(da)
```

Output:

```
DYN_ARR Size/Cap: 1/4 [1024]
18 32
```

```
DYN_ARR Size/Cap: 1/10 [1024]
```

Example #3:

```
da = DynamicArray()
print(da.length(), da.get_capacity())
[da.append(1) for i in range(100)]          # step 1 - add 100 elements
print(da.length(), da.get_capacity())
[da.remove_at_index(0) for i in range(68)]  # step 2 - remove 68 elements
print(da.length(), da.get_capacity())
da.remove_at_index(0)                       # step 3 - remove 1 element
print(da.length(), da.get_capacity())
da.remove_at_index(0)                       # step 4 - remove 1 element
print(da.length(), da.get_capacity())
[da.remove_at_index(0) for i in range(14)]  # step 5 - remove 14 elements
print(da.length(), da.get_capacity())
da.remove_at_index(0)                       # step 6 - remove 1 element
print(da.length(), da.get_capacity())
da.remove_at_index(0)                       # step 7 - remove 1 element
print(da.length(), da.get_capacity())

for i in range(14):
   print("Before remove_at_index(): ", da.length(), da.get_capacity(), end="")
   da.remove_at_index(0)
   print(" After remove_at_index(): ", da.length(), da.get_capacity())
```

Output:

```
0 4
100 128
32 128
31 128
30 62
16 62
15 62
14 30
Before remove_at_index():  14 30 After remove_at_index():  13 30
Before remove_at_index():  13 30 After remove_at_index():  12 30
Before remove_at_index():  12 30 After remove_at_index():  11 30
Before remove_at_index():  11 30 After remove_at_index():  10 30
Before remove_at_index():  10 30 After remove_at_index():  9 30
Before remove_at_index():  9 30 After remove_at_index():  8 30
Before remove_at_index():  8 30 After remove_at_index():  7 30
Before remove_at_index():  7 30 After remove_at_index():  6 14
Before remove_at_index():  6 14 After remove_at_index():  5 14
Before remove_at_index():  5 14 After remove_at_index():  4 14
Before remove_at_index():  4 14 After remove_at_index():  3 14
Before remove_at_index():  3 14 After remove_at_index():  2 10
Before remove_at_index():  2 10 After remove_at_index():  1 10
```

```
Before remove_at_index():  1 10 After remove_at_index():  0 10
```

Example #4:

```
da = DynamicArray([1, 2, 3, 4, 5])
print(da)
for _ in range(5):
    da.remove_at_index(0)
    print(da)
```

Output:

```
DYN_ARR Size/Cap: 5/8 [1, 2, 3, 4, 5]
DYN_ARR Size/Cap: 4/8 [2, 3, 4, 5]
DYN_ARR Size/Cap: 3/8 [3, 4, 5]
DYN_ARR Size/Cap: 2/8 [4, 5]
DYN_ARR Size/Cap: 1/8 [5]
DYN_ARR Size/Cap: 0/8 []
```

slice(self, start_index: int, size: int) -> object:

This method returns a new DynamicArray object that contains the requested number of elements from the original array, starting with the element located at the requested start index. If the array contains N elements, a valid `start_index` is in range [0, N - 1] inclusive. A valid size is a non-negative integer.

If the provided start index or size is invalid, or if there are not enough elements between the start index and the end of the array to make the slice of the requested size, this method raises a custom "DynamicArrayException". Code for the exception is provided in the skeleton file.

Example #1:

```
da = DynamicArray([1, 2, 3, 4, 5, 6, 7, 8, 9])
da_slice = da.slice(1, 3)
print(da, da_slice, sep="\n")
da_slice.remove_at_index(0)
print(da, da_slice, sep="\n")
```

Output:

```
DYN_ARR Size/Cap: 9/16 [1, 2, 3, 4, 5, 6, 7, 8, 9]
DYN_ARR Size/Cap: 3/4 [2, 3, 4]
DYN_ARR Size/Cap: 9/16 [1, 2, 3, 4, 5, 6, 7, 8, 9]
DYN_ARR Size/Cap: 2/4 [3, 4]
```

Example #2:

```
da = DynamicArray([10, 11, 12, 13, 14, 15, 16])
print("SOURCE:", da)
slices = [(0, 7), (-1, 7), (0, 8), (2, 3), (5, 0), (5, 3), (6, 1), (6, -1)]
for i, cnt in slices:
    print("Slice", i, "/", cnt, end="")
    try:
        print(" --- OK: ", da.slice(i, cnt))
    except:
        print(" --- exception occurred.")
```

Output:

```
SOURCE: DYN_ARR Size/Cap: 7/8 [10, 11, 12, 13, 14, 15, 16]
Slice 0 / 7 --- OK:  DYN_ARR Size/Cap: 7/8 [10, 11, 12, 13, 14, 15, 16]
Slice -1 / 7 --- exception occurred.
Slice 0 / 8 --- exception occurred.
Slice 2 / 3 --- OK:  DYN_ARR Size/Cap: 3/4 [12, 13, 14]
Slice 5 / 0 --- OK:  DYN_ARR Size/Cap: 0/4 []
Slice 5 / 3 --- exception occurred.
Slice 6 / 1 --- OK:  DYN_ARR Size/Cap: 1/4 [16]
Slice 6 / -1 --- exception occurred.
```

merge(self, second_da: object) -> None:

This method takes another DynamicArray object as a parameter, and appends all elements from this array onto the current one, in the same order in which they are stored in the input array.

Example #1:

```
da = DynamicArray([1, 2, 3, 4, 5])
da2 = DynamicArray([10, 11, 12, 13])
print(da)
da.merge(da2)
print(da)
```

Output:

```
DYN_ARR Size/Cap: 5/8 [1, 2, 3, 4, 5]
DYN_ARR Size/Cap: 9/16 [1, 2, 3, 4, 5, 10, 11, 12, 13]
```

Example #2:

```
da = DynamicArray([1, 2, 3])
da2 = DynamicArray()
da3 = DynamicArray()
da.merge(da2)
print(da)
da2.merge(da3)
print(da2)
da3.merge(da)
print(da3)
```

Output:

```
DYN_ARR Size/Cap: 3/4 [1, 2, 3]
DYN_ARR Size/Cap: 0/4 []
DYN_ARR Size/Cap: 3/4 [1, 2, 3]
```

map(self, map_func) ->object:

This method creates a new dynamic array where the value of each element is derived by applying a given `map_func` to the corresponding value from the original array.

This method works similarly to the Python `map()` function. For a review on how Python's `map()` works, here is some suggested reading:

- https://docs.python.org/3/library/functions.html#map
- https://www.geeksforgeeks.org/python-map-function/

Example #1:

```
da = DynamicArray([1, 5, 10, 15, 20, 25])
print(da)
print(da.map(lambda x: (x ** 2)))
```

Output:

```
DYN_ARR Size/Cap: 6/8 [1, 5, 10, 15, 20, 25]
DYN_ARR Size/Cap: 6/8 [1, 25, 100, 225, 400, 625]
```

Example #2:

```
def double(value):
    return value * 2

def square(value):
    return value ** 2

def cube(value):
    return value ** 3

def plus_one(value):
    return value + 1

da = DynamicArray([plus_one, double, square, cube])
for value in [1, 10, 20]:
    print(da.map(lambda x: x(value)))
```

Output:

```
DYN_ARR Size/Cap: 4/4 [2, 2, 1, 1]
DYN_ARR Size/Cap: 4/4 [11, 20, 100, 1000]
DYN_ARR Size/Cap: 4/4 [21, 40, 400, 8000]
```

filter(self, filter_func) ->object:

This method creates a new dynamic array populated only with those elements from the original array for which `filter_func` returns True.

This method works similarly to the Python `filter()` function. For a review on how Python's `filter()` works, here is some suggested reading:

- https://docs.python.org/3/library/functions.html#filter
- https://www.geeksforgeeks.org/filter-in-python/

Example #1:

```
def filter_a(e):
    return e > 10

da = DynamicArray([1, 5, 10, 15, 20, 25])
print(da)
result = da.filter(filter_a)
print(result)
print(da.filter(lambda x: (10 <= x <= 20)))
```

Output:

```
DYN_ARR Size/Cap: 6/8 [1, 5, 10, 15, 20, 25]
DYN_ARR Size/Cap: 3/4 [15, 20, 25]
DYN_ARR Size/Cap: 3/4 [10, 15, 20]
```

Example #2:

```
def is_long_word(word, length):
    return len(word) > length

da = DynamicArray("This is a sentence with some long words".split())
print(da)
for length in [3, 4, 7]:
    print(da.filter(lambda word: is_long_word(word, length)))
```

Output:

```
DYN_ARR Size/Cap: 8/8 [This, is, a, sentence, with, some, long, words]
DYN_ARR Size/Cap: 6/8 [This, sentence, with, some, long, words]
DYN_ARR Size/Cap: 2/4 [sentence, words]
DYN_ARR Size/Cap: 1/4 [sentence]
```

reduce(self, reduce_func, initializer=None) ->object:

This method sequentially applies the `reduce_func` to all elements of the dynamic array and returns the resulting value. It takes an optional initializer parameter. If this parameter is not provided, the first value in the array is used as the initializer. If the dynamic array is empty, the method returns the value of the initializer (or `None`, if one was not provided).

This method works similarly to the Python `reduce()` function. For a review on how Python's `reduce()` works, here is some suggested reading:

- https://www.geeksforgeeks.org/reduce-in-python/

Example #1:

```
values = [100, 5, 10, 15, 20, 25]
da = DynamicArray(values)
print(da)
print(da.reduce(lambda x, y: (x // 5 + y ** 2)))
print(da.reduce(lambda x, y: (x + y ** 2), -1))
```

Output:

```
DYN_ARR Size/Cap: 6/8 [100, 5, 10, 15, 20, 25]
714
11374
```

Explanation:

45 = (100 // 5) + 5^2
109 = (45 // 5) + 10^2
246 = (109 // 5) + 15^2
449 = (246 // 5) + 20^2
714 = (449 // 5) + 25^2

 # -1 is the initializer
11374 = -1 + 100^2+ 5^2 + 10^2 + 15^2 + 20^2 + 25^2

Example #2:

```
da = DynamicArray([100])
print(da.reduce(lambda x, y: x + y ** 2))
print(da.reduce(lambda x, y: x + y ** 2, -1))
da.remove_at_index(0)
print(da.reduce(lambda x, y: x + y ** 2))
print(da.reduce(lambda x, y: x + y ** 2, -1))
```

Output:

```
100
9999
None
-1
```

find_mode(arr: DynamicArray) -> (DynamicArray, int):

Write a standalone function outside of the DynamicArray class that receives a dynamic array already in sorted order, either non-descending or non-ascending. The function will return a tuple containing (in this order) a dynamic array comprising the mode (most-occurring) value/s of the array, and an integer that represents the highest frequency (how many times they appear).

If there is more than one value that has the highest frequency, all values at that frequency should be included in the array being returned in the order in which they appear in the input array. If there is only one mode, only that value should be included.

You may assume that the input array will contain at least one element, and that values stored in the array are all of the same type (either all numbers, or strings, or custom objects, but never a mix of these). You do not need to write checks for these conditions.

For full credit, the function must be implemented with O(N) complexity with no additional data structures (beyond the array you return) being created.

Example #1:

```
test_cases = (
    [1, 1, 2, 3, 3, 4],
    [1, 2, 3, 4, 5],
    ["Apple", "Banana", "Banana", "Carrot", "Carrot", "Date", "Date", "Date",
     "Eggplant", "Eggplant", "Eggplant", "Fig", "Fig", "Grape"]
)

for case in test_cases:
    da = DynamicArray(case)
    mode, frequency = find_mode(da)
    print(f"{da}\nMode: {mode}, Frequency: {frequency}\n")

case = [4, 3, 3, 2, 2, 2, 1, 1, 1, 1]
da = DynamicArray()
for x in range(len(case)):
    da.append(case[x])
    mode, frequency = find_mode(da)
    print(f"{da}\nMode: {mode}, Frequency: {frequency}\n")
```

Output:

```
DYN_ARR Size/Cap: 6/8 [1, 1, 2, 3, 3, 4]
Mode: DYN_ARR Size/Cap: 2/4 [1, 3], Frequency: 2

DYN_ARR Size/Cap: 5/8 [1, 2, 3, 4, 5]
Mode: DYN_ARR Size/Cap: 5/8 [1, 2, 3, 4, 5], Frequency: 1
```

```
DYN_ARR Size/Cap: 14/16 [Apple, Banana, Banana, Carrot, Carrot, Date, Date,
Date, Eggplant, Eggplant, Eggplant, Fig, Fig, Grape]
Mode: DYN_ARR Size/Cap: 2/4 [Date, Eggplant], Frequency: 3

DYN_ARR Size/Cap: 1/4 [4]
Mode: DYN_ARR Size/Cap: 1/4 [4], Frequency: 1
DYN_ARR Size/Cap: 2/4 [4, 3]
Mode: DYN_ARR Size/Cap: 2/4 [4, 3], Frequency: 1
DYN_ARR Size/Cap: 3/4 [4, 3, 3]
Mode: DYN_ARR Size/Cap: 1/4 [3], Frequency: 2
DYN_ARR Size/Cap: 4/4 [4, 3, 3, 2]
Mode: DYN_ARR Size/Cap: 1/4 [3], Frequency: 2
DYN_ARR Size/Cap: 5/8 [4, 3, 3, 2, 2]
Mode: DYN_ARR Size/Cap: 2/4 [3, 2], Frequency: 2
DYN_ARR Size/Cap: 6/8 [4, 3, 3, 2, 2, 2]
Mode: DYN_ARR Size/Cap: 1/4 [2], Frequency: 3
DYN_ARR Size/Cap: 7/8 [4, 3, 3, 2, 2, 2, 1]
Mode: DYN_ARR Size/Cap: 1/4 [2], Frequency: 3
DYN_ARR Size/Cap: 8/8 [4, 3, 3, 2, 2, 2, 1, 1]
Mode: DYN_ARR Size/Cap: 1/4 [2], Frequency: 3
DYN_ARR Size/Cap: 9/16 [4, 3, 3, 2, 2, 2, 1, 1, 1]
Mode: DYN_ARR Size/Cap: 2/4 [2, 1], Frequency: 3
DYN_ARR Size/Cap: 10/16 [4, 3, 3, 2, 2, 2, 1, 1, 1, 1]
Mode: DYN_ARR Size/Cap: 1/4 [1], Frequency: 4
```

Part 2 - Summary and Specific Instructions

1. Implement a Bag ADT class by completing the skeleton code provided in the file `bag_da.py`. You will use the DynamicArray class that you implemented in Part 1 of this assignment as the underlying data storage for your Bag ADT.

2. Once completed, your implementation will include the following methods:

   ```
   add()
   remove()
   count()
   clear()
   equal()
   __iter__()
   __next__()
   ```

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 Bag at any given time will be between 0 and 1,000,000 inclusive. The bag must allow for the storage of duplicate objects.

5. RESTRICTIONS: You are NOT allowed to use ANY built-in Python data structures and/or their methods in any of your solutions. This includes built-in Python lists, dictionaries, or anything else. You must solve this portion of the assignment by importing and using objects of the DynamicArray class (that you wrote in Part 1) and using class methods to write your solution.

   You are also **not** allowed to directly access any variables of the DynamicArray class (e.g. `self._size`, `self._capacity`, and `self._data`). Access to DynamicArray variables must only be done by using the DynamicArray class methods.

   Read the *Coding Guides and Tips* module for a detailed description of these topics.

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

This method adds a new element to the bag. It must be implemented with O(1) amortized runtime complexity.

Example #1:

```
bag = Bag()
print(bag)
values = [10, 20, 30, 10, 20, 30]
for value in values:
    bag.add(value)
print(bag)
```

Output:

```
BAG: 0 elements. []
BAG: 6 elements. [10, 20, 30, 10, 20, 30]
```

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

This method removes any one element from the bag that matches the provided `value` object. It returns `True` if some object was actually removed from the bag. Otherwise, it returns `False`. This method must be implemented with O(N) runtime complexity.

Example #1:

```
bag = Bag([1, 2, 3, 1, 2, 3, 1, 2, 3])
print(bag)
print(bag.remove(7), bag)
print(bag.remove(3), bag)
print(bag.remove(3), bag)
print(bag.remove(3), bag)
print(bag.remove(3), bag)
```

Output:

```
BAG: 9 elements. [1, 2, 3, 1, 2, 3, 1, 2, 3]
False BAG: 9 elements. [1, 2, 3, 1, 2, 3, 1, 2, 3]
True BAG: 8 elements. [1, 2, 1, 2, 3, 1, 2, 3]
True BAG: 7 elements. [1, 2, 1, 2, 1, 2, 3]
True BAG: 6 elements. [1, 2, 1, 2, 1, 2]
False BAG: 6 elements. [1, 2, 1, 2, 1, 2]
```

count(self, value: object) -> int:

This method returns the number of elements in the bag that match the provided `value` object. It must be implemented with O(N) runtime complexity.

Example #1:

```
bag = Bag([1, 2, 3, 1, 2, 2])
print(bag, bag.count(1), bag.count(2), bag.count(3), bag.count(4))
```

Output:

```
BAG: 6 elements. [1, 2, 3, 1, 2, 2] 2 3 1 0
```

clear(self) -> None:

This method clears the contents of the bag. It must be implemented with O(1) runtime complexity.

Example #1:

```
bag = Bag([1, 2, 3, 1, 2, 3])
print(bag)
bag.clear()
print(bag)
```

Output:

```
BAG: 6 elements. [1, 2, 3, 1, 2, 3]
BAG: 0 elements. []
```

equal(self, second_bag: “Bag”) -> bool:

This method compares the contents of a bag with the contents of a second bag provided as a parameter. The method returns `True` if the bags are equal (contain the same number of elements, and also contain the same elements without regard to the order of elements). Otherwise, it returns `False`. An empty bag is only considered equal to another empty bag.

This method must not change the contents of either bag. You are allowed to directly access all instance variables of `second_bag`, but you may not create any additional data structures, nor sort either bag. The runtime complexity of this implementation should be no greater than $O(N^2)$. The maximum test case size for this method will be limited to bags of 1,000 items each.

Example #1:

```
bag1 = Bag([10, 20, 30, 40, 50, 60])
bag2 = Bag([60, 50, 40, 30, 20, 10])
bag3 = Bag([10, 20, 30, 40, 50])
bag_empty = Bag()

print(bag1, bag2, bag3, bag_empty, sep="\n")
print(bag1.equal(bag2), bag2.equal(bag1))
print(bag1.equal(bag3), bag3.equal(bag1))
print(bag2.equal(bag3), bag3.equal(bag2))
print(bag1.equal(bag_empty), bag_empty.equal(bag1))
print(bag_empty.equal(bag_empty))
print(bag1, bag2, bag3, bag_empty, sep="\n")

bag1 = Bag([100, 200, 300, 200])
bag2 = Bag([100, 200, 30, 100])
print(bag1.equal(bag2))
```

Output:

```
BAG: 6 elements. [10, 20, 30, 40, 50, 60]
BAG: 6 elements. [60, 50, 40, 30, 20, 10]
BAG: 5 elements. [10, 20, 30, 40, 50]
BAG: 0 elements. []
True True
False False
False False
False False
True
BAG: 6 elements. [10, 20, 30, 40, 50, 60]
BAG: 6 elements. [60, 50, 40, 30, 20, 10]
BAG: 5 elements. [10, 20, 30, 40, 50]
BAG: 0 elements. []
False
```

`__iter__()`:

This method enables the Bag to iterate across itself. Implement this method in a similar way to the example in the Exploration: Encapsulation and Iterators.

You **ARE** permitted (and will need to) initialize a variable to track the iterator's progress through the Bag's contents.

You can use either of the two models demonstrated in the Exploration - you can build the iterator functionality inside the Bag, or you can create a separate iterator class.

Example #1:

```
bag = Bag([5, 4, -8, 7, 10])
print(bag)
for item in bag:
    print(item)
```

Output:

```
BAG: 5 elements. [5, 4, -8, 7, 10]
5
4
-8
7
10
```

`__next__()`:

This method will return the next item in the Bag, based on the current location of the iterator. Implement this method in a similar way to the example in the Exploration: Encapsulation and Iterators.

Example #2:

```
bag = Bag(["orange", "apple", "pizza", "ice cream"])
print(bag)
for item in bag:
    print(item)
```

Output:

```
BAG: 4 elements. [orange, apple, pizza, ice cream]
orange
apple
pizza
ice cream
```