Programming Assignment 3

ECS 032B FQ 2022

Due: December 4th, 2022

(Hard Deadline: December 8th, 2022 at 6PM)

Setup:

Python3.6+ to develop and run your code.

Provided files:

```
PA3 /
        __init__.py
        hashtable.py
        bst.py
        sortAndSearch.py
        heaps.py
        data.csv
        queries.json
        test.py
        tests/
                test_hashtable.py
                test_bst.py
                test_sortAndSearch.py
                test_heaps.py
```

Submission instructions:

Submit all provided files through Gradescope as a zip file.

Do not remove/rename files nor change any naming conventions.

You should receive your grade for correctness automatically.

Rubric:

Correctness = 18 points (Up to +6 extra points)

Section 1: +9 points

Section 2: +9 points

Section 3: +9 points

Section 4: +3 points

You only need to do 2 of the 3 sections (1,2,3).

Additional section and Section 4 are meant for extra credit (Up to +6 points)

Comments = 2 points

Your comments should include analysis of time/space complexities.

Extra Credit (Maximum: 2 points)

+0.4 points for every day you turn in all parts early.

Section 1: Hash Tables (hashtable.py)

You may use the random library and any built-in Python functionalities.
You may modify or add in other class variables and methods if you wish.

    Parent class is `DynamicArray` (see utils.py)
    You should resize the internal structure whenever necessary.
    You must write your own `reallocate` function.

For collision resolutions, we will use open addressing, with three different probing methods available.
Make appropriate design considerations for each probing method.

```
class HashTable(DynamicArray):
        def __init__(self, size:int, probe = 0:int):
```

Constructor for a hash table.

The types of probing available are {0: linear, 1: quadratic, 2: random}

Limit random probing to probe $\leq$ 1000 slots.

```
        def hashCode(self, key: int) -> int:
```

Return the hash code H for `key`.

$$H = \text{data} \bmod \text{size}$$

```
        def __getitem__(self, key: int) -> Any:
```

Retrieve the value for the `key`

```
        def __setitem__(self, key: int, value: Any) -> None:
```

Add a key-value pair to the hash table.

Overwrite the value if the key already exists.

Collisions should be resolved using the appropriate probing method.

```
        def __delitem__(self, key: int) -> None:
```

Remove the key-value pair using `key`.

```
        def loadfactor(self) -> float:
```

Return the current load factor of the hash table.

Section 2: Binary Search Tree (bst.py)

You need to make the appropriate changes to enforce the BST property.

```
class BST(BinaryTree):
        def __init__(self, arr: list, sort = False: bool):
```

Constructor for a binary search tree.

The parent class is `BinaryTree` (see utils.py).

Insert value in `arr` into the binary search tree.

If `sort` is True, then insert the values in `arr` in a specified order such that you have a balanced tree. (See below)

```
def addNode(self, data: Any) -> None:
```

Adds a node `data` to the tree tree using BST property.

```
def removeNode(self, data: Any) -> None:
```

Remove the node `data` from the tree using BST property.

```
def search(self, data: Any) -> bool:
```

Checks whether `data` is in the tree using BST property.

```
def tolist(self) -> list:
```

Return the binary search tree in a form of a sorted list.

```
def height(self, node) -> int:
```

Return the height of subtree with `node` as the root.

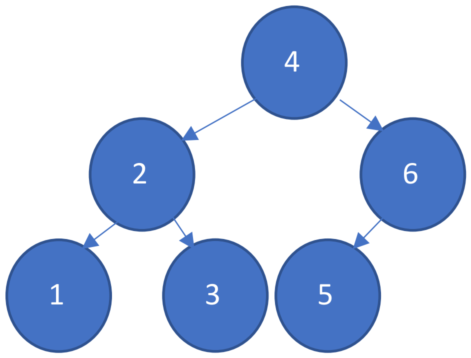
```
def balancefactor(self, node) -> int:
```

Return the balance factor of `node` in the tree.

<u>LIST <-> BST (sort = True)</u>

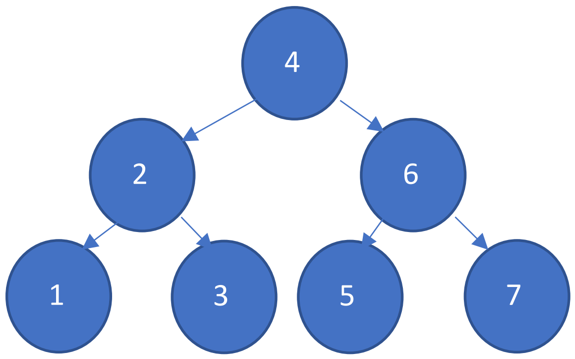
Some possible inputs for tree on the right:

[1, 2, 3, 4, 5, 6] (The output of tolist())
[2, 1, 3, 6, 4, 5]
[6, 5, 4, 3, 2, 1]
…



Some possible inputs for tree on the right:

[1, 2, 3, 4, 5, 6, 7] (The output of tolist())
[3, 2, 1, 7, 6, 5, 4]
[7, 4, 5, 2, 1, 3, 6]
…



Section 3: Sort and Search (sortAndSearch.py)

In this section, sort and search on a large collection of crude oil price data from 1987-2022. We do not provide any starter code for this section. See comments on the top of the file.

Write your code in the main function that runs the sort and search.

1. Implement and benchmark linear search using the unsorted data (data.csv) on the queries of prices (queries.json).
2. Implement and benchmark BubbleSort on (data.csv) by the prices in ascending order.
3. Implement and benchmark one other sorting algorithm on (data.csv) by the prices in ascending order. This should run faster than BubbleSort.
4. Implement and benchmark another searching algorithm that assumes a sorted collection on (query.json). This should run faster than linear search.

Benchmark all algorithms in seconds.

In the main function, generate the following output file:

- output.csv
    - In this file, you store the following information:
        - Execution time of linear search.
        - Output of linear search.
            - List of [Date:str, Price:str] in order of queries
        - Execution time of BubbleSort.
        - Output of BubbleSort.
            - List of [Date:str, Price:str] in ascending prices.
        - Execution time of other sorting algorithm.
        - Output of other sorting algorithm.
        - Execution time of other searching algorithm.
            - List of [Date:str, Price:str] in ascending prices.
        - Output of other searching algorithm.
            - List of [Date:str, Price:str] in order of queries

The tester will only run your main function.

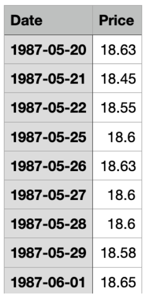
You should perform all logic in the main function.

The tester for this section will take up to 2-3 minutes. (See test.py to set checkpoints)

You are allowed to use any additional built-in modules you want.

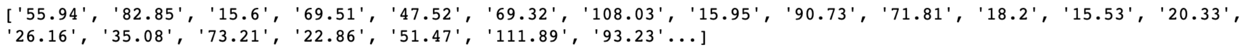
To help, some modules that will be helpful:

- time (https://docs.python.org/3/library/time.html)
    - Use time.time() to measure wall-clock time.
- To load/modify the csv file:
    - csv (https://docs.python.org/3/library/csv.html)



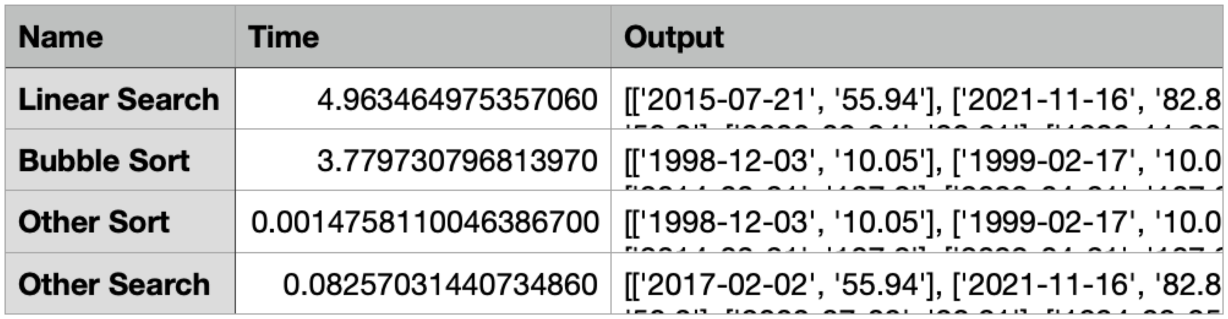
Date	Price
1987-05-20	18.63
1987-05-21	18.45
1987-05-22	18.55
1987-05-25	18.6
1987-05-26	18.63
1987-05-27	18.6
1987-05-28	18.6
1987-05-29	18.58
1987-06-01	18.65

Partial screenshot of data.csv



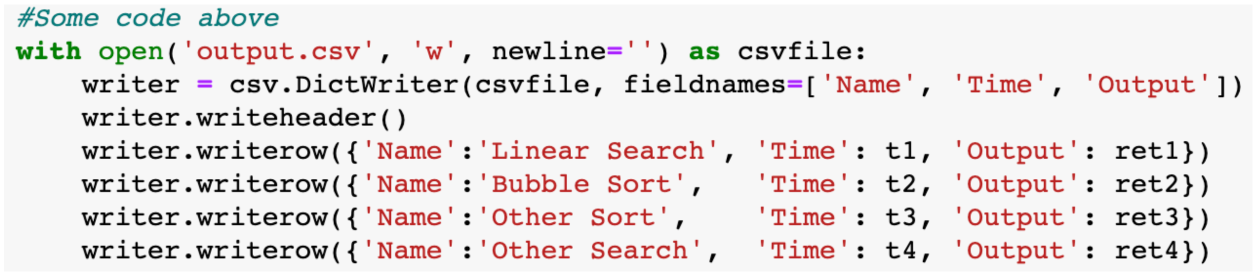
```
['55.94', '82.85', '15.6', '69.51', '47.52', '69.32', '108.03', '15.95', '90.73', '71.81', '18.2', '15.53', '20.33',
'26.16', '35.08', '73.21', '22.86', '51.47', '111.89', '93.23'...]
```

Partial screenshot of queries.json



Name	Time	Output
Linear Search	4.963464975357060	[['2015-07-21', '55.94'], ['2021-11-16', '82.8
Bubble Sort	3.779730796813970	[['1998-12-03', '10.05'], ['1999-02-17', '10.0
Other Sort	0.0014758110046386700	[['1998-12-03', '10.05'], ['1999-02-17', '10.0
Other Search	0.08257031440734860	[['2017-02-02', '55.94'], ['2021-11-16', '82.8

Partial screenshot of output.csv.
Keep the names of the column and row headers the same.



```
#Some code above
with open('output.csv', 'w', newline='') as csvfile:
    writer = csv.DictWriter(csvfile, fieldnames=['Name', 'Time', 'Output'])
    writer.writeheader()
    writer.writerow({'Name':'Linear Search', 'Time': t1, 'Output': ret1})
    writer.writerow({'Name':'Bubble Sort',   'Time': t2, 'Output': ret2})
    writer.writerow({'Name':'Other Sort',    'Time': t3, 'Output': ret3})
    writer.writerow({'Name':'Other Search',  'Time': t4, 'Output': ret4})
```

A hint on how to write your output.csv.

Section 4: Heaps (heap.py)

You may use the heapq library (https://docs.python.org/3/library/heapq.html#heapq.heapify)

```
def heapify(arr:list, isMax:bool) -> BinaryTree
```

This function will form a heap from the `arr`, assume to be level-order representation.
`isMax` will determine whether your heap is a max-heap or min-heap.
Return the heap as a `BinaryTree` (see utils.py)