Python: Exploring Data Structures

In this python set there will be various topics covering different types of data structures. Their purpose will be utilizing the basics of the language itself to produce expected outputs based on given inputs. There will likely be different folders with built in test-case scripts along with the respective inputs. I will point out anything you need in order to be able to run the test suite. I also provide some solutions of my own for each section.

In order to use your classes in others, use: *from <file> import * — see below*

-=-

Before beginning, here are a few tips on inheritance and polymorphism in python:

When defining classes, you usually have the `__init__()` function which acts like a constructor would in another OOPL. Unlike those, however, python explicitly requires you to pass in the ‘`self`’ keyword.

For example,

```
class SimpleClass:
    def __init__(self):
        self.var1 = 0
        self.var2 = 1
```

Our simple class has two instance variables now defined. The ‘`self`’ keyword indicates that the method is to operate on the given object. In a language like Java, the keyword would be ‘`this`’, but there it is *implicitly* passed. Usually you would only need it when differentiating a shadowed variable with its instance variable counterpart. To inherit from this class, you would use it as follows:

```
from SimpleClass import *:

class SubClass(SimpleClass):
    def __init__(self):
        super().__init__()
```

1. To start with something simple, create a new python script called *ListNode.py* in the folder named ‘LinkedList’. This file will contain a node class with three members and three methods. The members and their respective methods will be:

I. data:	`getData()`
II. previous:	`getPrevious()`
III. next:	`getNext()`

An approach in Java might be:

```
public class Node <T> {

     private Node<T> prev;
     private Node<T> next;
     private T data;

     public Node(T data)
     {
          this.data = data;
          this.prev = null;
          this.next = null;
     }
     ...
}
```

After designing the node class, create another class called *List*, in another script called *List.py*. This will, much like java, serve as a base class for things such as a doubly linked list and a regular linked list. In this class, you will have the list operations as follows (Keep in mind the capital 'L' so as not to invoke the built in list data structure) :

```
clear() - Removes all of the elements. The list
hereafter should be reset to its empty state.

get(index) - Traversing the list, you should get the
node at this specified index. If it is empty, then
return None. The node should be "peeked" so NOT
removed. As with arrays and lists, if the specified
index is out of range, throw an error. The indexing
will work as a normal array.

contains(obj) - Same as get only it returns True or
False if the object is in the list. To clarify, the
data should be examined to check if there is a match.

getSize() - Returns the current size of the list.

getHead() - Returns the head of the list.

getTail() - Returns the tail of the list.

printList() - Prints the list's contents in order, from
the head to the tail of the list. The format must be as
follows:
         data
         data
         data

in other words, only the data per node, per line.
If it is empty, then return an empty string. For
testing purposes, place each data element in an array
and return it.
```

Lastly comes the implementing of the classes which inherit from this one. Going in order, you will have these two files: *LinkedList.py* and *DoublyLinkedList.py.*

The LinkedList file implementation is described below.

insert(obj, index=-1) - Insert object at the specified index. If the index is -1, then by default it will place items at the tail of the list. Otherwise, it will place them at the index, moving what's there over by one. Additionally, a node will be created within this method to hold your data. As with arrays and lists, if the specified index is out of range, throw an error. This ONLY applies if it's not the index referring to what would be the end of the list.

remove(obj, index=-1) - If the index isn't -1, then ONLY remove the object at the specified index and ignore the parameter. If, by default, the index is left as -1, then it will search the list for this data. You are free to implement the search how you wish, but remember that you already have other methods that perform this for you. The data itself should be returned. If it isn't found then it will return None. It should be noted that if left as -1 and the data is not found, then None will also be returned. This behavior is expected with accessing an empty list. As with arrays and lists, if the specified index is out of range, throw an error.

Note: In order to not break the program, for negative ranges check for *less than -1.* This will allow all default behaviors described.

DoublyLinkedList will perform the exact same operations as above. The only difference that you have to take into account is the 'previous' member. You will add one more method, however, to print the list in reverse:

printListReverse() - Prints the list in **reverse** order to demonstrate the capability of a doubly linked list. The format must be as follows:

```
data
data
data
```

in other words, only the data per node, per line. If it is empty, then return an empty string. For testing purposes, place each data element in an array and return it.

We could simulate the use of interfaces by placing the shared methods between the two list classes in the main superclass as well and using the ‘`pass`’ keyword on them.

To test your code, locate the file *ListTests.py* and simply run it. Make any adjustments as needed. ***Make sure your code is in the same directory!*

Linked Lists Analysis:

Linked lists are similar in operation to that of an array. Consider, though, that with an array, the size must be generally known ahead of time and the usage of memory therefore becomes a valuable asset. As data grows arbitrarily, an array may not be efficient; if you have an array whose size is used up and wish to add another element, then doing so is expensive as it would require not only creating a new array, but copying each element one-by-one.

Benefits of an array:

- Contiguous in memory (each element is side-by-side), so iteration is simple.

- Indexing is the primary advantage, so accessing an element is O(1).

- Because it is contiguous, it is cache-friendly in terms of locality.

Cons of an array:

- Not dynamic, so it is unable to easily grow and shrink.

- Size must generally be guessed ahead of time.

- Adding an extra element beyond the size of the array is expensive and not efficient. Doing so could result in an O(n) algorithm for copying elements and usage of extra memory for the new array.

- Removing an element may be efficient if you do so in-place, but you will have an element in the array with which to mark as “unusable” temporarily. Usually, this is instead done by removing the element and then shifting each element over, which is not as efficient.

Benefits of a Linked list:

- Dynamic data structure, and can grow and shrink as necessary; memory usage is limited to an as-needed basis.

- Insertion and deletion will always be a constant-time operation. Unlike an array, you are able to place and delete an element wherever you choose within a list. The only drawback is that inserting or deleting anywhere other than the front or the back, will usually be O(n) due to having to search or reach the element desired.

- As stated above, searching the list can be O(1) *best case.* For example: suppose you are using an insertion sort algorithm (sort occurs as you insert within the list) and later decide to search for a max/min element. Depending on how the list is sorted, this will result in an instant search; this can also be done with an array, but insertion sort is more difficult with an array as compared with a list.

Cons of a Linked list:

- As the data structure is dynamic, the elements are NOT contiguous in memory. This means that unlike an array, lists cannot be indexed numerically - at least not in a necessarily efficient way. The least efficient way of doing so is a linear search O(n), and the most efficient way of doing so is O(log n).

- Relying on system memory allocation. In C, you have to manually free any memory that you requested from the system. Failure in doing so has the possibility of causing memory leaks. Of course now it’s relatively rare to run out of system memory for your programs.

- Although it may be convenient to have a dynamic data structure and use memory on an as-needed basis, the construction of a single object or even a structure in C can be more expensive compared to a single array element.

Why Big-O and Asymptotic Runtime Matters

Earlier I mentioned O(n) and O(log n). Among those are many other runtimes applicable to countless algorithms. At first they may not matter much as the data you work with can be ridiculously tiny at best. Consider, however, when that data set grows to millions, even billions, of entries.

In the folder titled *Big-O_Demo* you'll find a simple python script called *binary_search.py* (ignore *in.txt)*. Within it you'll find two algorithms:

- linear search
- binary search

After generating all 30 million numbers, the program will store them in an array and search through them using these search algorithms. For all intents and purposes, suppose you didn't know the size of the data set - in this case, the array is sorted already for binary search. Now after waiting a little bit of time you'll be prompted to enter a number to search for. I recommend searching for a number relatively high in the list so as to force the algorithm to take longer in its search. Once the linear search has completed, the same number you entered will be used to drive binary search. You'll notice that the binary search completed much faster than linear. For me the difference was vast: 6.625 seconds for linear search and 0.000021 seconds for binary search.

The reason behind this is that unlike linear search which essentially has to iterate through the entire data set to reach a number (worst case), binary search only has to look through a fraction of the data set for each iteration; this is known as a *divide and conquer* algorithm. Effectively, the array is "cut" in half and the search space is dwindled down until no more remains. As the data set increases dramatically, algorithms like this help reduce the time it takes to work with it, making the program overall more efficient.

For perspective, the runtimes have been listed.

Linear search best case: O(1)
Linear search worst case: O(n)

Binary search best case: O(1)
Binary search worst case: O(log n)

Note that for binary search to work, the data set has to be sorted, which is another algorithmic problem of its own.

Queues

A variety of implementations exists for queues. We will take a look at 3 different ones in the following order: *Queue, PriorityQueue, Heap*. For ease of use and fairness, we will use the built-in python list.

The test cases for the PQ's will consist of a list of names in a file corresponding to a unique priority. All the names are guaranteed to be unique. The goal will be to not only implement the queues, but time them and see how they perform with relatively large sets of data, much like the Big-O_Demo. For both the priority queue implementations, we will be using a max priority queue, ordered from highest to lowest priority.

In the *Queues* folder, start by creating a script called *Queue.py.*

According to the python documentation, to enable the use of a queue, we must import the *deque* module as follows: *from collections import deque*
See the python *list* or *collections* documentation on using this module.

As this is simply a normal queue, it follows a FIFO ruleset. The methods and their functionality are described below.

It's important to note that this file and all others will be classes just as before. Most if not all methods will simply be wrappers that utilize existing methods from the deque module.

`enqueue(obj) - Inserts the given object to the end of the queue/list. This operation should be O(1) with the given implementation.`

`dequeue() - Removes the object at the beginning of the queue/list. This should also be O(1).`

`isEmpty() - Returns True or False if the queue is empty.`

`size() - Returns the current size of the queue.`

`printQueue() - Simply print the queue as you would a list. For testing purposes, return the queue itself. Do note that if you don't convert it to a list in this method, it will return` *`deque(…)`*

This is the most simple implementation of a queue. Unsorted, but following the rules of a first-in, first-out structure: The first elements to arrive will be the first to leave, much like the line analogies. These queues will later be used as well.

Queues are great, and widely used, but what about fairness? Consider another line analogy, but this time in the context of a hospital: the patients to arrive with the most critical health conditions will be of highest priority, and those with minor health conditions will be of lower priority; nonetheless, this still follows a queue-like structure, as the highest priority patients will be treated first, and the lowest priority patients will be treated last.

When we talk about priority queues, the "priority" can be based on various factors, the simplest being maximum priority queues and minimum priority queues. For our purposes, we will be exploring this data structure using maximum priority, going from highest to lowest. As will be seen later, priority queues can be made more efficient using heaps.

Earlier I mentioned how sorting was an algorithmic problem of its own. In fact, many algorithms exist to make the problem of sorting more efficient. Python uses the *Timsort* sorting algorithm, which is a hybrid of insertion sort and merge sort. For more info on Timsort, see: https://en.wikipedia.org/wiki/Timsort

For the primary priority queue implementations (not the heap), there will be three versions of enqueue. The first version is the same as the Queue.py version, so it will be **inherited**. The second version will be an insertion sort and the third will be using python's built-in sorting algorithm. Overall, they will be timed to see which implementation is the best one.

Additional Notes:

Insertion sort itself is an O(n) worst case algorithm, but with n elements to insert, the total runtime is O(n^2) worst. O(1) best case leads to total of O(n).

Timsort itself is O(n) best case, but with n elements to insert, the total runtime is O(n^2). Its worst case runtime is O($nlogn$), which with n elements to insert leads to a total of O(n^2logn). To be accurate, we aren't considering the "already sorted" best case scenario.

In a new script called *PriorityQueue.py* start by setting up the inheritance of the Queue class. Then implement the methods described below.

——— overridden methods ———

enqueue(obj, priority) - Inserts the given object to the end of the queue/list. This operation should be O(1) with the given implementation. Inserting the element should be as a tuple in the following format: (obj, priority)

dequeue() - Removes the object *anywhere* in the list with the highest priority. Should any priorities be the same, only dequeue the one that occurs first.

---

enqueue_insert(obj, priority) - Inserts the given object in the queue using insertion sort. The object should be a tuple using the same format as described above; that being said, you search for the correct spot in the queue to insert based on the priority, moving highest to lowest. ALSO NOTE: If the priority adjacent is the same, then stop searching and insert at the current spot (aka the element will be placed before the adjacent one).

enqueue_timsort(obj, priority) - Inserts the given object in the queue using the built-in python timsort algorithm; that is, using *.sort()* on the queue. This will be done for every enqueue operation. This is the most important part: Since you are sorting the tuples based on priority, you must first do: ***from operator import**** *. From that point you must convert the queue to a temporary list in order to then do: ***.sort(key=itemgetter(1), reverse=True)***
Failure to do this will result in alphabetical sorting.

dequeue_priority() - Removes the object in the list with the *highest* priority. Note that at this point, the highest priority element should be at the front of the list. This assumption will be made when testing.

Concept: Heaps

As a data structure, heaps have many uses. In some cases, we think of heaps in the context of dynamic memory allocation or object allocation in OOPL's. Other applications include: sorting, PQ implementation, graph algorithms, and more.

We will be exploring the use of heaps in implementing a more efficient version of the priority queue. Noting the current tactics we are using, the priority queue as-is has varying runtimes for *enqueue, enqueue_insert, enqueue_timsort;* as a result, the *dequeue* and *dequeue_priority* operations are affected. Below is some analysis in the worst-case scenario:

-=-

`enqueue` (regular) - No modifications to this method except for supporting the *priority* parameter. Enqueueing an object is therefore O(1) since it goes immediately to the end of the list. With n elements to insert, this yields a total O(n) runtime.

`dequeue` (With regular enqueue) - Since we are using this operation to extract the object with the highest priority, we have to search the entire queue as it is currently unsorted. This results in a worst-case runtime of O(n).

-=-

`enqueue_insert` - This method utilizes insertion sort. We've established that insertion sort in its worst-case is O(n) since it has to search for the correct spot to place. With n elements to insert, the runtime overall is O(n^2).

`dequeue_priority` (After insertion sort) - We are still dequeueing based off of highest priority. This method makes the assumption that this element is now at the front of the queue. Since we've sorted, this should be true. The runtime for this operation is O(1).

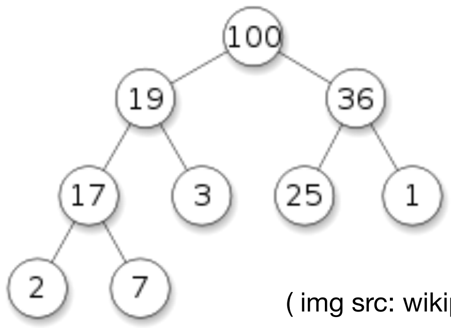
-=-

`enqueue_timsort` - This method inserts the object to the back of the queue and then performs the built-in python Timsort algorithm. The worst-case is O($nlogn$) and with n elements to insert, the overall is O(n^2logn). What you'll notice if you graph these functions, n^2logn eventually supersedes n^2, so insertion sort here is actually more efficient.

`dequeue_priority` (After Timsort) - Same as above.

So, why are heaps the better implementation? We'll get into brief analysis in a bit. First, let's take a look at the primary ideas of a heap.

The heaps we will be using are also referred to as *binary heaps*, which take on a binary tree-like structure. This simply means that every node has at most two children. Typically heaps are drawn as a tree-like structure, but are implemented using arrays:



(img src: wikipedia)

[0] 100	[1] 19	[2] 36	[3] 17	[4] 3	[5] 25	[6] 1	[7] 2	[8] 7	[9] etc

Specifically, here is shown a *max heap*. For a max-heap the largest element is always the root (aka the first element), and each child is less than or equal to that of the parent. The property to access the parent, left, and right node is simple:

parent: $\lfloor \frac{i-1}{2} \rfloor$ (zero indexed) $\lfloor \frac{i}{2} \rfloor$ (not starting at zero)
left: $2i + 1$ (zero indexed) $2i$ (not starting at zero)
right: $2i + 2$ (zero indexed) $2i + 1$ (not starting at zero)

Maintaining a max- or min-heap property utilizes the following rules for the parent-child relationships. This is more formal than I described above:

MAX HEAP: left/right node <= parent node
(The largest of the two children, if bigger than the parent will be swapped to be the new parent node)

MIN HEAP: left/right node >= parent node
(The smallest of the two children, if smaller than the parent will be swapped to be the new parent node)