

# 9.1 - Week 9 - Applied - Theory

---

## Objectives of this Applied Session


## Objectives of this Applied Session

- To understand the utility and syntax of iterators and generators
- To be able to compare linked and array based implementations of ADTs covered previously.

---

# Recap on ADT implementations

For each of the following Datatype implementations, briefly described how they work, making particular note of where on the datatype certain operations are performed (front/back).

 A diagram on paper/whiteboard works well for these questions.

## Question 1

The StackADT implemented with a Linked structure.

*No response*

## Question 2

The StackADT implemented with an Array structure.

*No response*

## Question 3

The QueueADT implemented with a Linked structure.

*No response*

## Question 4

The QueueADT implemented with a CircularQueue Array structure.

*No response*

---

# Comparing implementations

Now that we've recapped what each implementation is, let's compare the usability of each implementation under certain problems.

**Question 1** *Submitted Sep 20th 2022 at 8:27:34 am*

Which methods of the StackADT would be affected if we changed the Array implementation to keep the top of the stack at the front of the array, rather than the end?

Analyse what these new complexities would be.

☒ push

☒ pop

☐ \_\_len\_\_

**Question 2** *Submitted Sep 20th 2022 at 8:41:28 am*

In passing, the following pros and cons of using a linked list vs an array have been discussed:

- Pro for array: Constant time access to any element.
- Pro for array: Easy/Fast traversal backwards.
- Pro for linked: No shuffling required.
- Pro for linked: Easy/Fast resizing (both up and down).

Identifying which of these Pros actually matter when comparing the efficiencies of the StackADT when implemented with Linked structures or the Array. Your answer should reference the runtime complexities of the two implementations methods.

☐ Constant time access to any element.

☐ Easy/Fast traversal backwards.

☐ No shuffling required.

☒ Easy/Fast resizing (both up and down).

### Question 3

Why does a Circular Queue need to be Circular? What limitations would the Queue have if we simply resized the underlying array when `front` got too large?

*No response*

### Question 4

In all previous content, we've studied Stacks and Queues side-by-side, since they have very similar properties. Why then, is there no `CircularStack`?

*No response*

---

# Debugging

Enough quizing for now, let's write some code! (Or at least read it 😊)

I've been tasked with writing an implementation for the StackADT method `sum_all`, which should:

1. Calculate the sum of all elements in a stack
2. Not modify the stack in the process

I've had 3 cracks at the implementation, but to no avail:

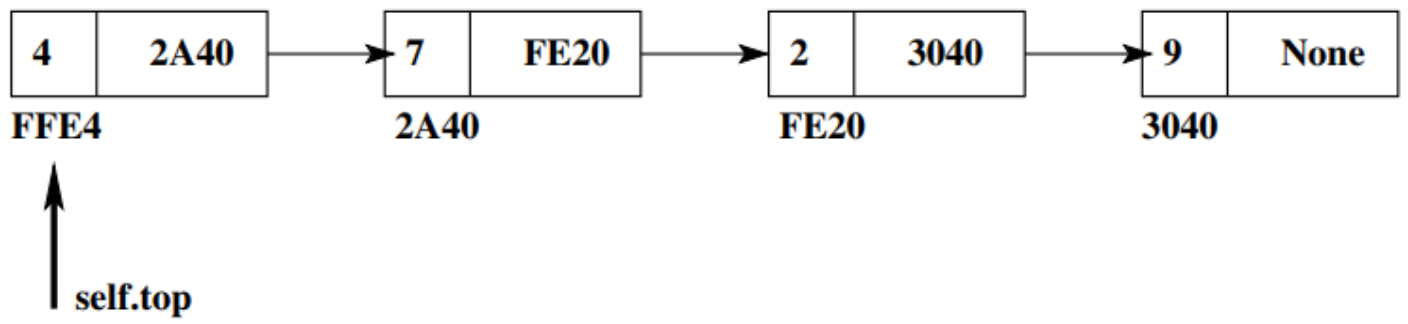
```
def sum_all(self) -> int:
    "implementation 1"
    total = 0
    while (not self.is_empty()):
        total += self.top.item
        self.top = self.top.link
    return total

def sum_all(self) -> int:
    "implementation 2"
    current = self.top
    total = 0
    while current.link is not None:
        total += current.item
        current = current.link
    return total

def sum_all(self) -> int:
    "implementation 3"
    current = self.top
    total = 0
    while current is not None:
        current = current.link
        total += current.item
    return total
```

## Task 1:

In `sum_all.txt`, explain why each of the 3 implementations is wrong. You may use the following example to demonstrate this:



### Task 2:

In `sum_all.py`, give your own implementation which correctly solves the problem.

---

## FIFO vs. LIFOLIFO

In `analysis.txt` (but preferably also on paper), Implement that QueueADT class using only 2 instances of StackADT. Describe the predicted worst-case complexity of your serve/append operations, as well as what your predicted average-case complexity would be for both operations.

*Note: average-case complexity is not something that will be examined/is expected to be properly understood in this unit, it is just mentioned as foreshadowing for future units.*

**Hint 1 (For coming up with the idea):**

► Expand

**Hint 2 (For analysing average case):**

► Expand

---

# Iterator recap

Let's again recap the content on iterators and generators.

**Question 1** *Submitted Sep 20th 2022 at 9:33:40 am*

What methods must an `Iterator` class implement?

☒ `__iter__`

☐ `__len__`

☒ `__init__`

☒ `__next__`

☐ `next`

☐ `finished`

**Question 2**

Since python allows us to, why shouldn't we make the ADT class also its own Iterator class? (Adding the `__iter__`, `__next__` methods to `StackADT`, rather than making a `StackADTIterator`, for example)

Try to give two examples where this might cause issues.

*No response*



---

## Delete Negatives

In `negatives.py`, write the python method of the `ListADT` class `delete_negative` which will delete all the negative values from a list. Assume you know nothing about the particular implementation used for the list (could be array or nodes). To be able to do so, you can assume the list class is iterable, that is, there already exists an `__iter__` method that returns an iterator for the current list, and the returned iterator has the following methods implemented:

- **has\_next** (is there a next element)
- **peek** (what is the next element)
- **\_\_next\_\_** (return next element, move forward)
- **delete** (delete next element)
- **reset** (reset iterator to the beginning of the list)

You should also state your predicted complexity of this implementation, and compare & contrast it to an implementation that does not use an iterator.

---

# Counting Primes

This unit is about Data Structures and Algorithms, so we mostly deal with iterable data structures and the iterators that work over their data. However, an iterator doesn't necessarily have to work over a data structure. This exercise is an example of an iterator that isn't backed by a container data structure. The Miller Rabin algorithm is a probabilistic algorithm for establishing whether a given integer is prime or composite. For now there's no need to know about the actual algorithm as we'll just be using it as a black box.

Define an iterator class `Primes` whose instances yields all the primes in sequence, one by one, as shown in the `__main__` section of code.