7.0 - Week 7 - Workshop (MA)

Learning objectives

- Understanding Queue ADT with arrays.
- Understanding List ADT with arrays.

Week 7 Padlet Discussion Board link: https://monashmalaysia.padlet.org/fermi/2022week7

How much do you know about Queues and Lists?

Question 1 *Submitted Sep 5th 2022 at 10:06:02 am*

What fundamental principle lies behind the Queue ADT?

● FIFO *(first in, first out)*

○ LIFO *(last in, first out)*

○ FILO *(first in, last out)*

○ LILO *(last in, last out)*

○ WTF *(way to fail..!)*

Question 2 *Submitted Sep 5th 2022 at 10:06:05 am*

What fundamental principle lies behind the List ADT?

○ FIFO *(first in, first out)*

○ LIFO *(last in, first out)*

○ FILO *(first in, last out)*

● NOTA *(none of the above)*

Question 3 *Submitted Sep 5th 2022 at 10:06:09 am*

What key parts of an *array-based* Queue do we need to have access to?

☑ `front`

□ `head`

□ `center`

□ `body`

☑ `rear`

□ `tail`

☑ `length`

Question 4 *Submitted Sep 5th 2022 at 10:06:19 am*

What key parts of an *array-based* List do we need to have access to?

□ `head`

□ `body`

□ `tail`

☑ `length`

□ `legs`

Question 5 *Submitted Sep 5th 2022 at 10:06:26 am*

Do we need to maintain the order of elements in a Queue / List?

○ No - queue, no - list

○ No - queue, yes - list

○ Yes - queue, no - list

● Yes - queue, yes - list

○ Why don't you just tell us?

Question 6 *Submitted Sep 5th 2022 at 10:06:40 am*

What is the advantage of a *Circular* Queue over a *Linear* Queue?

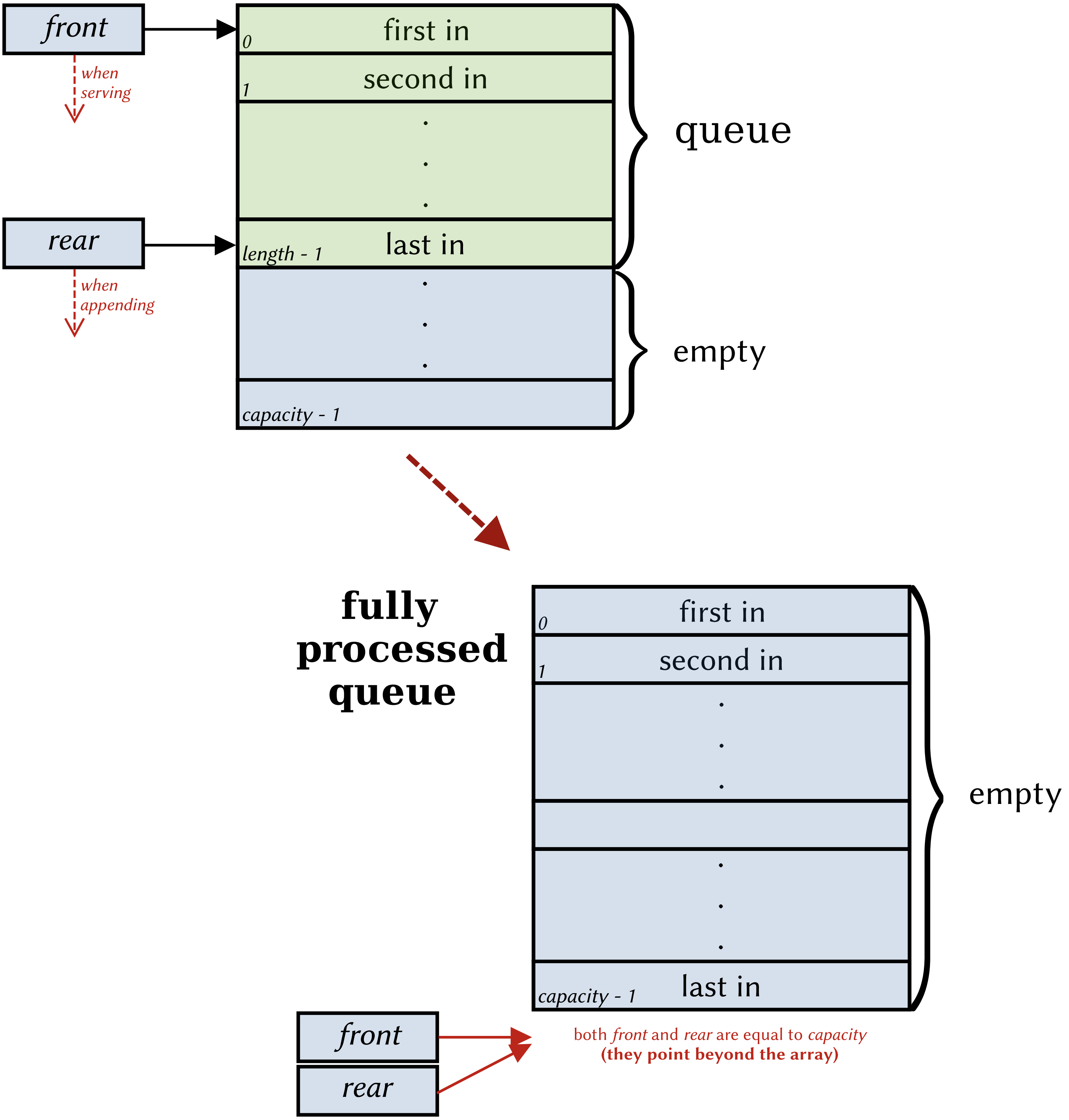
○ It has the shape of a circle and so looks nicer, visually!

● Linear array-based Queues can only be filled out once until they reach the capacity of the array. A Circular Queue can reuse previously occupied positions and so it is *the only true* array-based Queue!

○ Let's get to the next slide already!

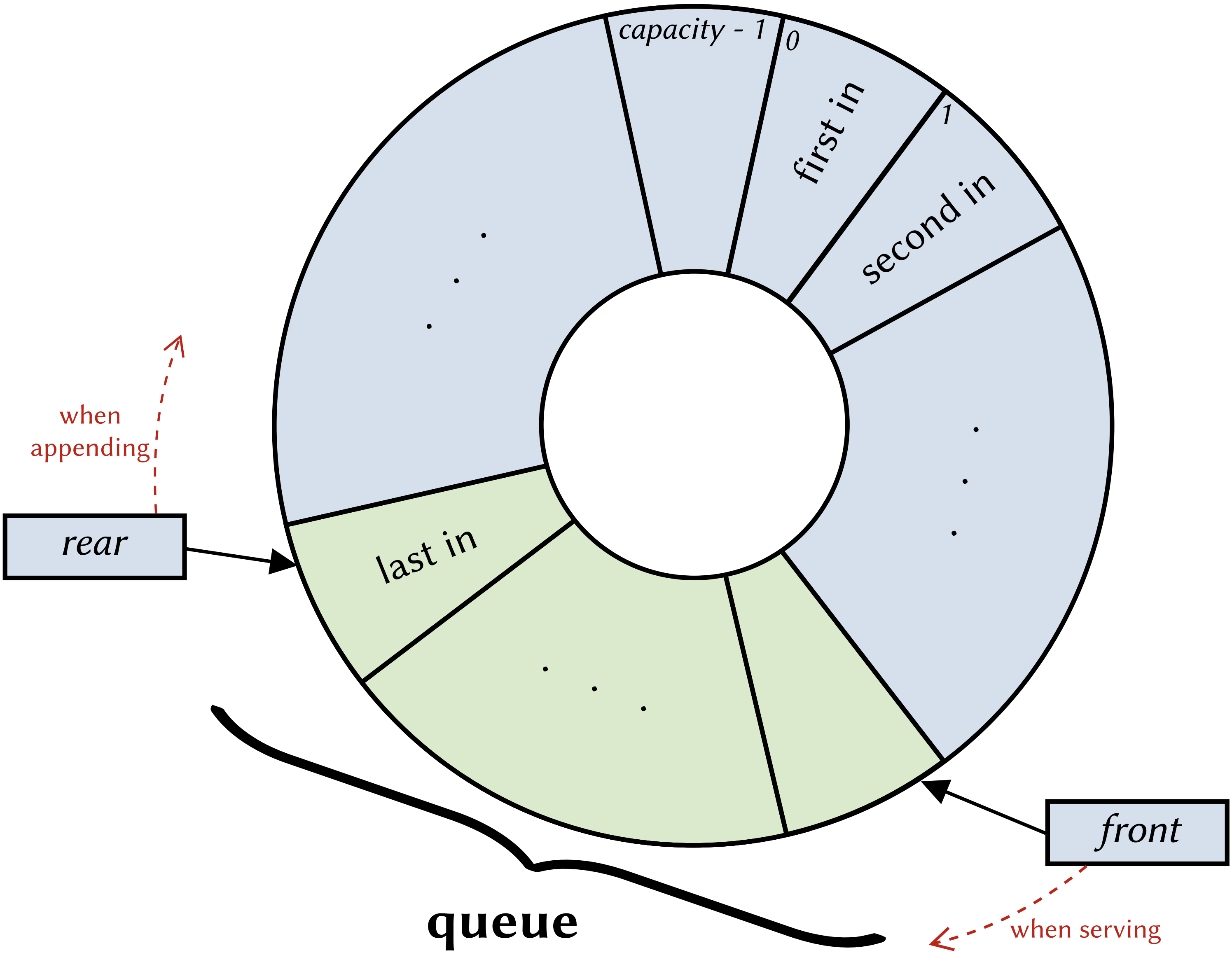
Linear Queues

✕ They can be effectively **used once** until they reach their full capacity.



Circular Queues

✓ There is **no such problem** with Circular Queues! Pointer `rear` is updated as `rear = (rear + 1) % capacity`.
Same applies to the `front` pointer! This way we are always within the bounds and can reuse all the cells!



Queue interface

```
""" Queue ADT. """

__author__ = 'Maria Garcia de la Banda modified by Alexey Ignatiev'
__docformat__ = 'reStructuredText'

from abc import ABC, abstractmethod
from typing import TypeVar, Generic

class Queue(ABC, Generic[T]):
    """ Abstract class for a generic Queue. """

    def __init__(self) -> None:
        """ Initialisation. """
        self.length = 0

    @abstractmethod
    def append(self,item:T) -> None:
        """ Adds an element to the rear of the queue."""
        pass

    @abstractmethod
    def serve(self) -> T:
        """ Deletes and returns the element at the queue's front."""
        pass

    @abstractmethod
    def is_full(self) -> bool:
        """ True if the stack is full and no element can be pushed. """
        pass

    def __len__(self) -> int:
        """ Returns the number of elements in the queue."""
        return self.length

    def is_empty(self) -> bool:
        """ True if the queue is empty. """
        return len(self) == 0

    def clear(self):
        """ Clears all elements from the queue. """
        self.length = 0
```

Implementing Circular Queues

The goal of this activity is to:

Given the abstract class `Queue` provided in the scaffold, complete the implementation of Circular Queues with arrays. In particular, implement the missing methods

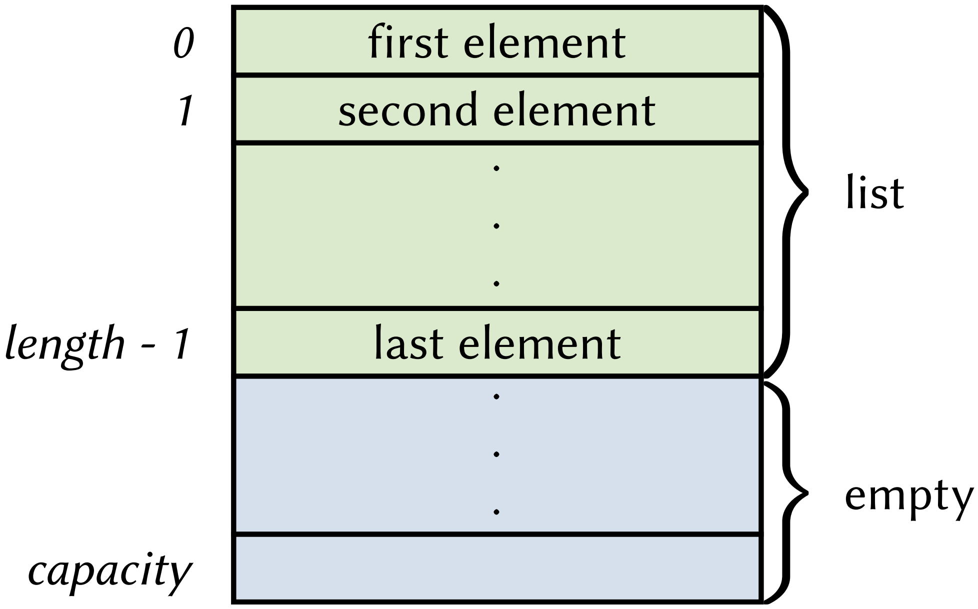
- `self.append()`
- `self.serve()`
- `self.is_full()`

✓ After you are done with the implementation, you can run it to see how it works (see the file `run_queue.py`).

✗ Next, analyse **time complexity** of all these methods on the *number of items* n in the queue.

Array-based Lists

It is simple to implement a list using an array. The elements of the list are compactly represented as the array's elements starting from position `0`.



Observe that we do not require any particular properties of lists except that
- the items of the list must be kept in order;
- we must have direct access to the first item;
- given any item, we must be able to access the next one.

Implementing Lists

The goal of this activity is to:

Given the abstract class `List` provided in the scaffold, complete the implementation of array-based Lists. In particular, implement the missing methods

- `self.index()`
- `self.delete_at_index()`
- `self.insert()`

✓ After you are done with the implementation, you can play with the implementation and to see how it works. For that, modify the file `run_list.py`.

✗ Next, analyse **time complexity** of all these methods given the *number of items* n in the list and the target `index` if provided. Think of when you need to take into account the *complexity of item comparison*.

Complexity of item removal

Question 1

Let the number of items in the list be n and the complexity of item comparison be $\mathcal{O}(c_=)$. Given the implementation of `self.index()` and `self.delete_at_index()` from the previous code challenge, what is the **best-case complexity** of `self.remove()` below? When is it achieved? Relate with the index of the item.

```
def remove(self, item: T) -> None:
    index = self.index(item)
    self.delete_at_index(index)
```

○ $\mathcal{O}(1)$, which happens when the item to remove is first

○ $\mathcal{O}(1)$, which happens when the item is last

○ $\mathcal{O}(n + c_=)$, which happens when the item is first

○ $\mathcal{O}(n \cdot c_=)$, which happens when the item is last

Question 2

Is there a simple way to reduce the best-case complexity of `remove()`?

○ True

○ False

Feedback Form

Weekly Workshop Feedback Form

Question 1

I am enrolled in:

○ □□ Australia

○ □□ Malaysia

Question 2

What needs improvement?

No response

Question 3

What worked best?

No response

Question 4

How engaged were you by the workshop?

○ □□□ Very engaged

○ □□□ Engaged

○ □😐□ Not impressed

○ 😕😴□ Lost