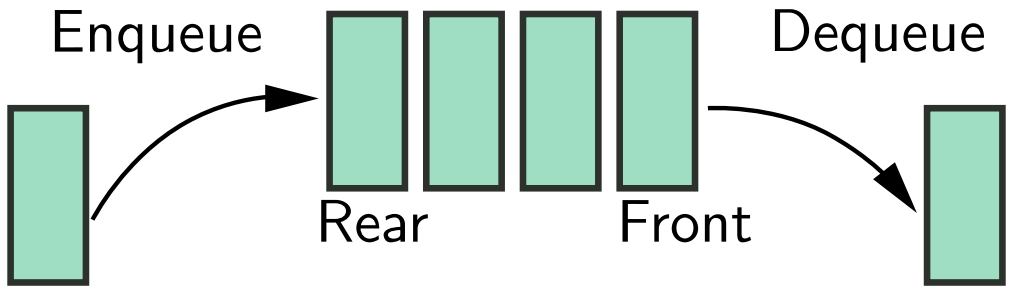
Queues

Queues = FIFOs (First-In-First-Out)

Queue is an abstract data type defined by its three operations:

- `enqueue(x)` puts value `x` at the *rear* of the queue
- `dequeue()` takes out a value from the *front* of the queue
  If there are no values in the queue, it raises `EmptyQueueException`.
- `is_empty()` says whether the queue is empty



Queue is an abstract data type. A queue is a list of values (e.g. integers, Booleans, ...). The two ends of the list are called the *rear* and *front*. We *enqueue* a value (add it to the rear of the queue) and *dequeue* a value (remove it from the front of the queue). Thus a queue behaves in a First In First Out (FIFO) manner. If we try to dequeue from an empty queue, we get an `EmptyQueueException`. In theory we should be able to enqueue any number of values, but in practice that won't be possible and we will get an exception if we run out of memory.

Example

Starting from an empty queue, enqueue 4 and 3, dequeue, then enqueue 1 and dequeue two times. What is the last value you get? What would happen in the next dequeue?

Usage

A typical application of queues is a print queue: files are sent to the queue for printing and are printed in the order in which they were sent.

After sending a file, you know that only the jobs currently in the queue will be printed before yours.

Queues are useful whenever we have need to process tasks in the order in which they came. We demonstrate this in the printer queue but there are many more examples (e.g. web server when serving websites).

Notice that since the tasks in a print queue are executed in the order in which they came, there is no priority. Even as a lecturer I have to wait for all student tasks that came before mine to finish before my file gets printed.

Queue ADT

Here is a possible list of operations for a Queue ADT (many variations are possible)[5]

Constructors and Accessors:

- `EmptyQueue`: returns an empty Queue
- `push(element, queue)`: (also called `enqueue`) pushes an element onto the back of the given queue.
- `top(queue)`: (also called `front`) returns the value at the front of the queue without changing the queue[6]
- `pop(queue)`: (also called `dequeue`) returns the queue with the front element removed[6]
- `isEmpty(queue)`: reports whether the queue is empty

[5] Read chapters 1 and 2 of the module handouts

[6] Triggers error if the queue is empty

Queue as a linked list

In order to have an efficient implementation we need to store the location of the last element in the linked list.

Remember: **Dequeue** from **front**, **enqueue** to **rear**

We have two options again:

1. Front at beginning of the linked list, rear at end
2. Rear at beginning of the linked list, front at end

Question: Which one is better and why?

How would enqueue and dequeue be implemented if we did (1) or (2)?

For (1) as long as we use 2 pointers in the head node, one to the first node of the linked list, one to the last node, it will take constant time to dequeue (remove from the start of the Linked List) and to enqueue (add a node at the end of the linked list).

For (2), again with 2 pointers in the head node, we can enqueue in constant time (insert a node at the start of the linked list). However, to dequeue, we now need the address of the penultimate node of the linked list in order to remove the last node, and to find that address, we will need to iterate through the whole linked list, costing linear time.

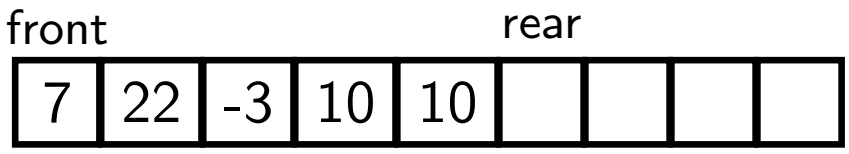
Thus we can ensure constant time enqueues and dequeues in case (1)

- `enqueue` = `insert_end`
- `dequeue` = `delete_beg`

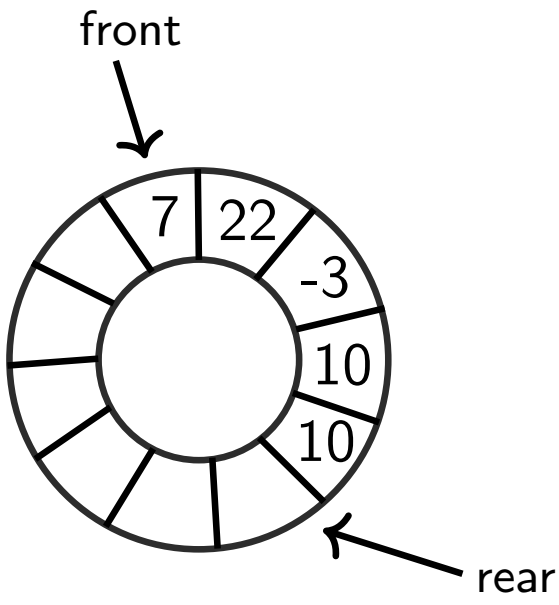
Case (2) gives us constant time enqueues but linear time dequeues.

Queue stored in an array (1st attempt)

Whenever we `enqueue` (resp. `dequeue`) we increment `rear` (resp. `front`):



We eventually run out of space! To fix this, every time we `dequeue`, move everything to the left. It works but is slow!



Instead we use a circular storage of a queue. We move Front and Rear clockwise.

Works beautifully in theory but how do we implement it in an array?

If we know that the size of queue is limited during the run of our program, we can store the queue as an array. We store the front position and size in variables `front` and `size`, respectively. Then, values stored in the queue are stored on the positions `front`, `front+1`, ..., `front+size-1`. We must maintain the invariant `front+size` $\leqslant$ `MAXQUEUE`.

To `enqueue` we store the new value in position `front+size` and increment `size`. To `dequeue` we read out the value on position `front` and increment `front`.

However, this way, we eventually reach the end of the array and we can't continue enqueueing elements even if there is space in the array to do so, that is, the space in the array where old values were which have since been dequeued.

We can fix this by, everytime we dequeue a value, copying all the elements in the queue down to the start of the array so that `front` is back to 0. But this makes dequeue very slow. There is a better solution using a *Circular Array Queue Implementation*

Digression: Invariants

An *invariant* is a condition on code. It must always be true during the execution of some section of code, e.g. a loop, or during the execution of a method, or, if it applies to a class, then it can be a condition that must be met by all objects of the class on every entry to and exit from the methods of the class even if it can be temporarily false during the execution of those methods.

Invariants are important because they specify conditions that must be met and maintained in parts of the code. So not only do they communicate information to the reader of the code, but they are tools that can be used both to identify and debug errors in the code (e.g. print an error message if this invariant is broken, stop in the debugger at any point when this invariant becomes false, etc.), and can be used to mathematically prove that the program is correct.

Digression: `div` and `mod` (maths break)

For numbers a, b with $b > 0$ we write `a div b` to mean the result of dividing a by b and discarding the remainder, and we write `a mod b` to mean the reminder.

For example: $123 \text{ div } 10 = 12$ $\quad$ $58.7 \text{ div } 10 = 5$

$123 \text{ mod } 10 = 3$ $\quad$ $58.7 \text{ mod } 10 = 8.7$

Example

Racing on a track: Assume that one lap in a race is 600 meters. Then a runner who has run 1550 meters has completed `1550 div 600 = 2` laps and `1550 mod 600 = 350` meters of the third lap.

Note that `a div b` is always an integer and $0 \leq$ `a mod b` $< b$ such that

$$(a \text{ div } b) * b + a \bmod b = a.$$

Consequently: $-7 \text{ div } 10 = -1$ and $-7 \bmod 10 = 3$

$-123 \text{ div } 10 = -13$ and $-123 \bmod 10 = 7$.

`mod` can be used to define the floor function (rounding down to an integer), we have

$$\lfloor x \rfloor = x \text{ div } 1 \quad \text{and} \quad x \text{ div } y = \lfloor x/y \rfloor$$

(very useful when we do complexity)

Note that in Java `Math.floorDiv` and `Math.floorMod` behave better than `%` and `/`, respectively, because the latter interpret the operations on negative numbers incorrectly (Java, and C, allows negative results for `mod`).

More examples:

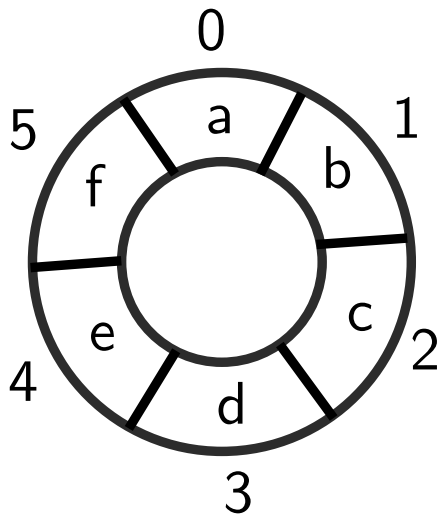
- $100 \bmod 10 = 0$ (100 is divisible by 10)
- $-18 \operatorname{div} 10 = -2$ (from 0, move backwards 2 * 10)
- $-18 \bmod 10 = 2$ (and then 2 steps forwards)

  (it has to be in the bounds: $0 \leq -18 \bmod 10 < 10$)

- Floor: $\lfloor 14.3 \rfloor = 14, \quad \lfloor 7.0 \rfloor = 7, \quad \lfloor -5.3 \rfloor = -6$
- Ceiling: $\lceil 14.3 \rceil = 15, \quad \lceil 7.0 \rceil = 7, \quad \lceil -5.3 \rceil = -5$

Representing circles

Positions in the circle, numbered clockwise, correspond to the positions in the array:



$\Longrightarrow$

0	1	2	3	4	5
a	b	c	d	e	f

Example

For a position `pos` in the circle (e.g. `pos = 5`), moving clockwise by one positions is computed as `(pos + 1) mod 6`.

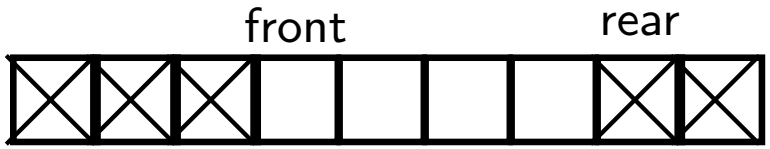
Or in general `(pos + 1) mod circle_length`.

Circular queue: Array implementation

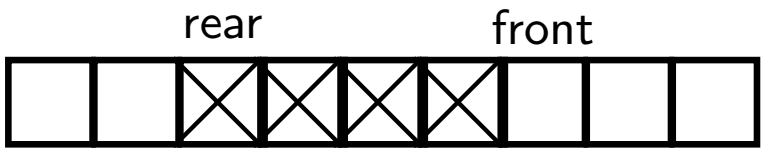
```
1 // Initialize an empty queue:
2 queue = new int[MAXQUEUE];
3 front = 0;
4 size = 0;
```

We store values in the queue in between positions marked by `rear` and `front`, that is,

- if $front + size \leqslant MAXQUEUE$ then the queue consists of the entries on positions $front, front + 1, ..., front + size - 1$:



- if $front + size > MAXQUEUE$ then the queue consists of the entries on positions $front, front + 1, ..., MAXQUEUE - 1$ and $0, 1, ..., (front + size - MAXQUEUE - 1)$:



Circular queue: Array implementation

The invariants maintained in this implementation are:

- $0 \leqslant front < MAXQUEUE$
- $0 \leqslant size \leq MAXQUEUE$

Note that rather than using a *rear* index variable, we will always calculate it when we need it from *front*, *size* and *MAXQUEUE*. Thus `rear = (first + size) mod MAXQUEUE`

The queue is full if `size == MAXQUEUE` and empty when `size == 0`

To enqueue, we first check that the queue is not full, put the new value at index position `(first + size) mod MAXQUEUE`, and increment *size* by adding 1 to it.

To dequeue, we first check that the queue is not empty, get the value at index position `front`, increment *front* by calculating `front = (front + 1) mod MAXQUEUE` and decrement *size* by subtracting 1 from it.

Circular queue: Array implementation

```
1 // Initialize empty queue:
2 queue = new int[MAXQUEUE];
3 front  = 0;
4 size = 0;
```

```
1 boolean is_empty () {
2   return size == 0;
3 }
```

```
1 boolean is_full () {
2   return size == MAXQUEUE;
3 }
```

```
1 void enqueue (int val) {
2   if (size == MAXQUEUE) { throw QueueFullException; }
3   queue[(front+size) mod MAXQUEUE] = val;
4   size ++;
5 }
```

```
1 int dequeue () {
2   int val;
3   if (size == 0) { throw QueueEmptyException; }
4   val = queue[front];
5   front = (front+1) mod MAXQUEUE
6   size --;
7   return val;
8 }
```

Double Ended Queues: Deques

While the Java library does have a `Stack<...>` class, it is an old design that has been kept for backwards compatibilty purposes and should not normally be used. For both `Stack` and `Queue` classes, you should use the `Deque<...>` class, which implements a *double-ended queue* data type. This has implementations `ArrayDeque<...>` and `LinkedList<...>` and supports inserting at and removing from both ends.

Actually, the `Deque<...>` class is really a Java *Interface*, rather than a full *Class*. This will be covered in your Java programming module but the distinction is not important for the purposes of this module.

Final points

As we will see, stacks and queues are used in many algorithms.

We often just say “make a stack” or “make a queue” and we don’t care how they are implemented.

We know that, whether it is as an array or linked list, it can be done efficiently.