Software Verification

Master Programme in Computer Science

Mário Pereira mjp.pereira@fct.unl.pt

Nova School of Science and Technology, Portugal

October 27, 2025

Lecture 8

based on previous editions by João Seco, Luís Caires, and Bernardo Toninho

also based on lectures by Andrei Paskevich and Claude Marché

1. Presentation of the second handout.

Verification of Imperative Programs (5/5)

2. ADTs and Typestates

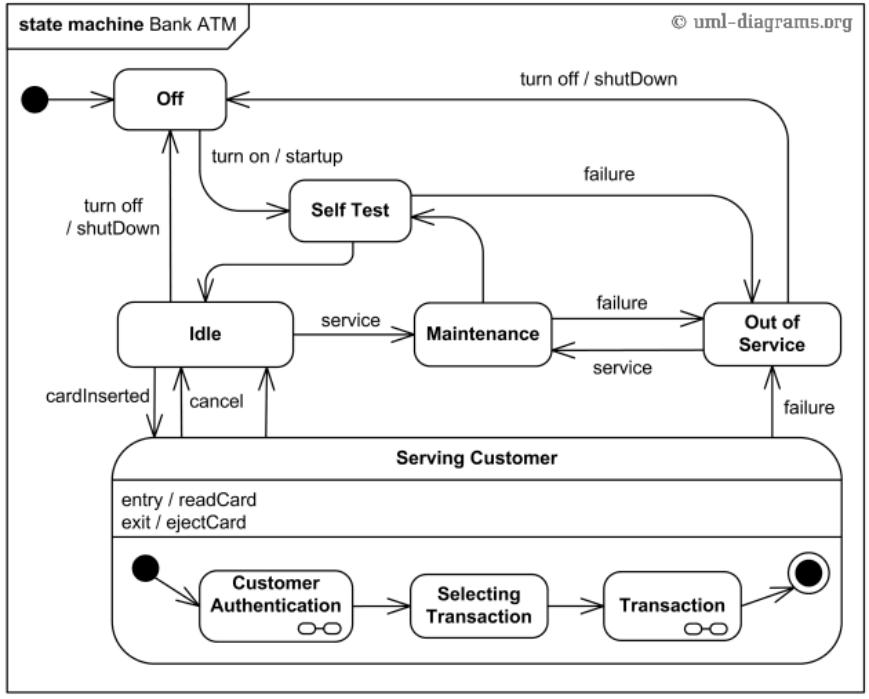
ADTs and Typestates

State Transition Diagrams

Typically the connection between a state and the domain of the values for an object are based on conventions written in documentations.

Operations are state transitions in a state diagram.

If a state is formally connected to conditions over the state of an object, the correction of state transitions may be mechanically checked.

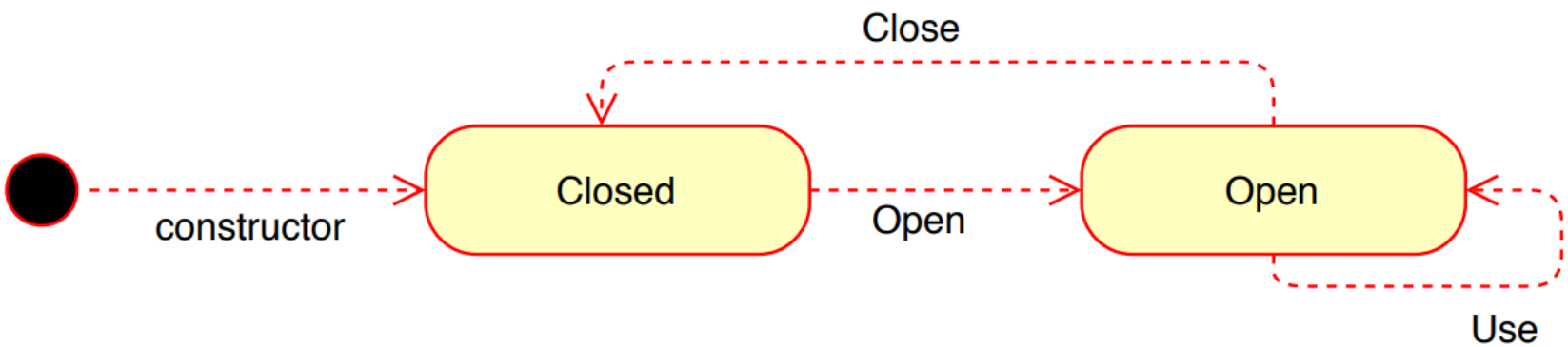


https://www.uml-diagrams.org/bank-atm-uml-state-machine-diagram-example.html?context=stm-examples

Typestates

In many situations, we may represent each abstract state of an ADT by a named assertion, that hides some set of concrete states.

We illustrate using a general Resource object with the following state diagram:



Typestates

In many situations, we may represent each abstract state of an ADT by a named assertion, that hides some set of concrete states.

We illustrate using a general Resource object with the following state diagram:

- A Resource must first be created and starts on the closed state
- A Resource can only be used after being Opened
- A Resource may be Closed at any time
- A Resource can only be Opened if it is in the Closed state, and Closed if it is in the Open state

We define two abstract states

- `closed_state`
- `open_state`

```
type t = {
  h: array int;
  size: int;
} invariant { 0 <= size <= h.length }

predicate open_state (t: t)
= ...

predicate closed_state (t: t)
= ...

let create ()
  ensures { closed_state result }
= ...
```

Typestates define an abstract layer, visible to clients that can be used to verify resource usage.

Protocol Insurance with Typestates

```
let using_the_resource ()
= let r = create () in
  open 2 r;
  use 2 r;
  use 9 r;
  close r
```

Legal usage of resource, according to protocol!

```
let using_the_resource ()
= let r = create () in
  open 2 r;
  use 2 r;
  use 9 r;
  close r
```

Illegal usage of resource, according to protocol!

```
type t = {
  h: array int;
  size: int;
} invariant { 0 <= size <= h.length }

predicate open_state (t: t)
= t.size = t.h.length

predicate closed_state (t: t)
= t.size = 0

let create ()
  ensures { closed_state result }
= { size = 0; h = make 0 0 }
```

Typestates define an abstract layer, that may be defined with relation to the type invariants and be used to verify the implementation.

```
type t = {
  h: array int;
  size: int;
} invariant { 0 <= size <= h.length }

let open (n: int) (t: t) : unit
  requires { n > 0 }
  requires { closed_state t }
  ensures  { open_state t }
= t.h <- make n 0;
  t.size <- n

let close (t: t) : unit
  requires { open_state t }
  ensures  { closed_state t }
= t.h <- make 0 0;
  t.size <- 0
```

Method implementations represent state transitions,

- must be implemented to correctly ensure the soundness
- of the arrival state
- assuming the departure state

Typestates – summary

In many situations, we may represent each abstract state of an ADT by a named assertion, that hides some set of concrete states.

It is often enough to expose Typestate assertions to ensure ADT soundness and no runtime errors.

In general, full functional specifications in terms the abstract state is too expensive and should be only adopted in high assurance code.

However, Typestate assertions are feasible and should be enforced in all ADTs.

The simplest Typestate is the type invariant (no variants/less specific).

Abstract Data Types – Key Points

Software Design Time

- Abstract Data Type
- What are the Abstract States / Concrete States?
- What is the Representation Invariant?
- What is the Abstraction Mapping?

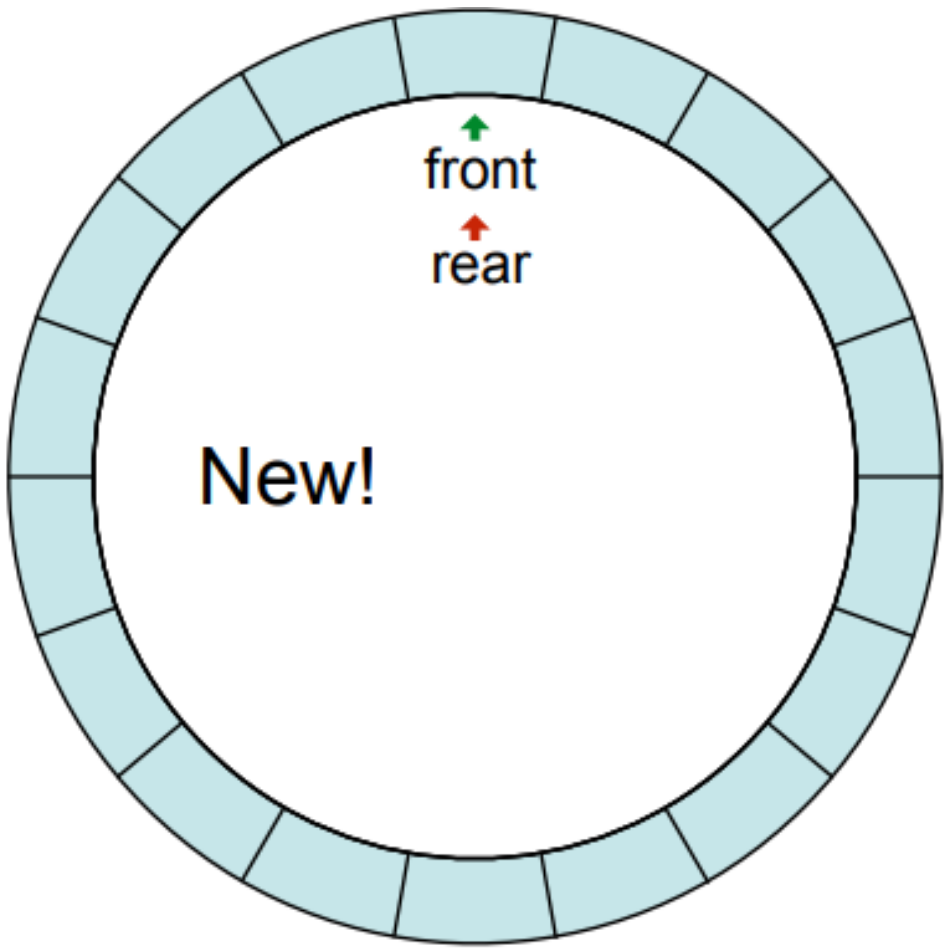
Software Construction Time

- Make sure creation function establishes the Type Invariant
- Make sure all operations preserve the Type Invariant
  - they may assume the Type Invariant
  - they may require extra pre-conditions (*e.g.*, on arguments)
  - they may enforce extra post-conditions

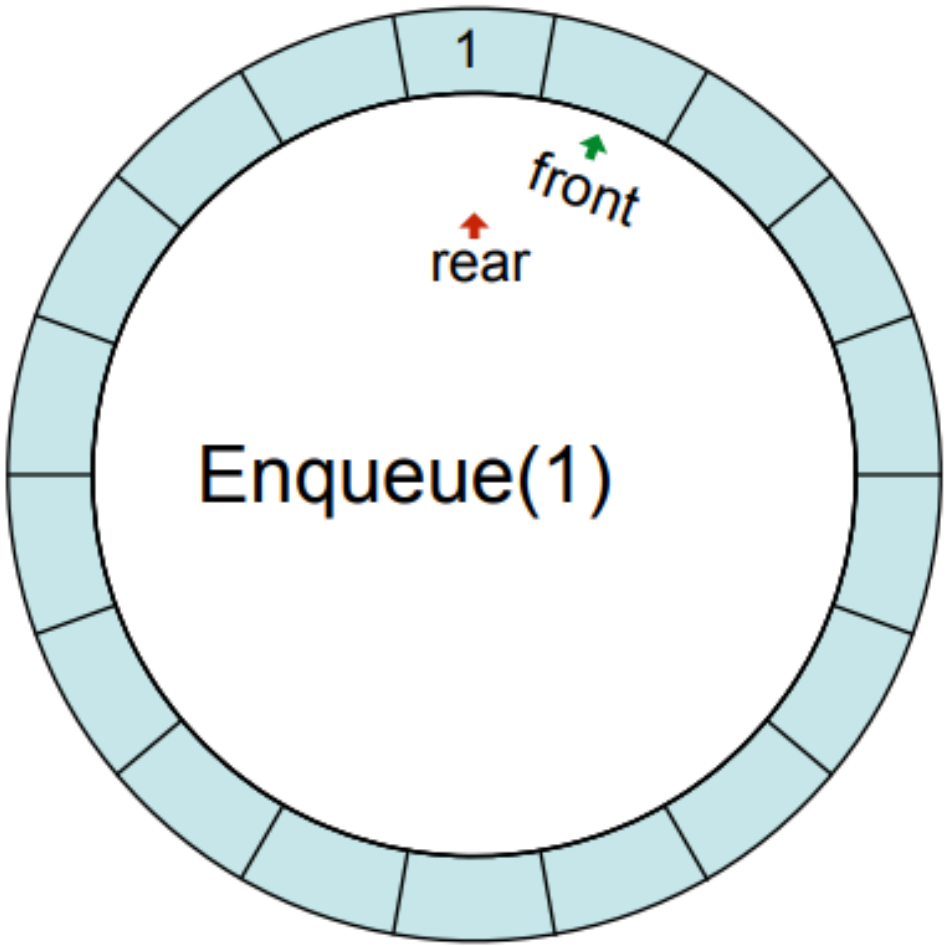
Use assertions and Typestates to make sure your ADT is sound.

Typestates – Case Study

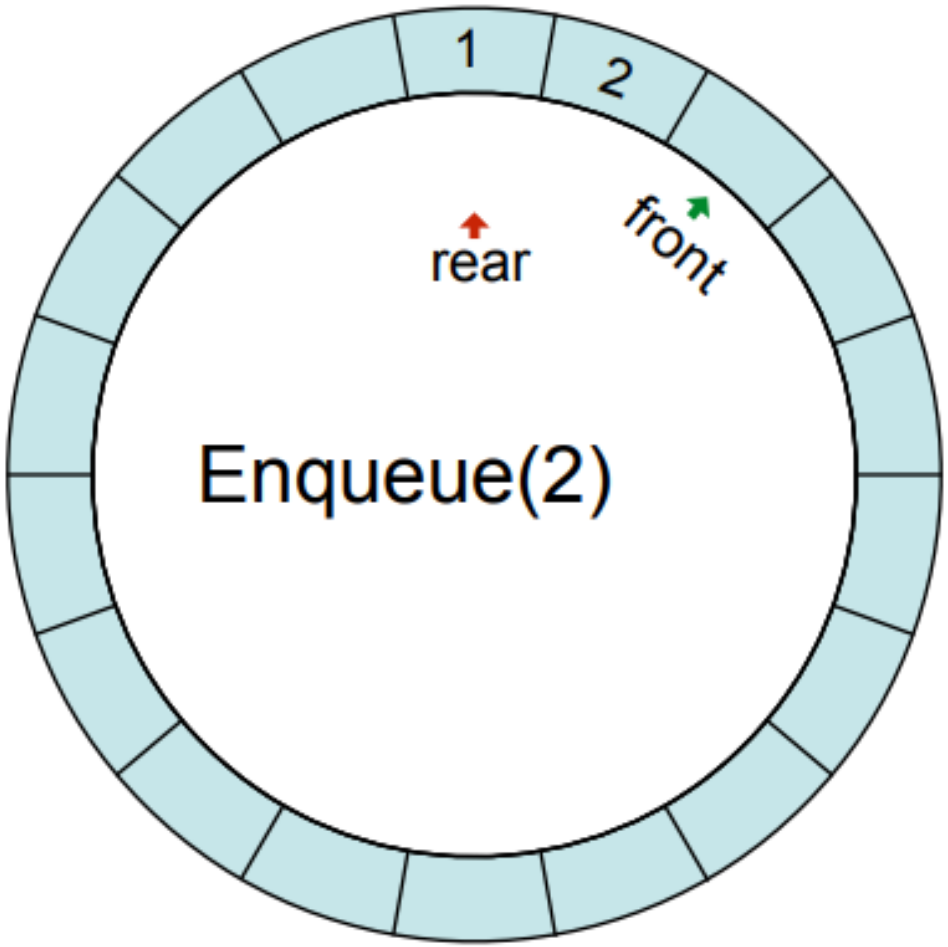
An implementation using a circular buffer.



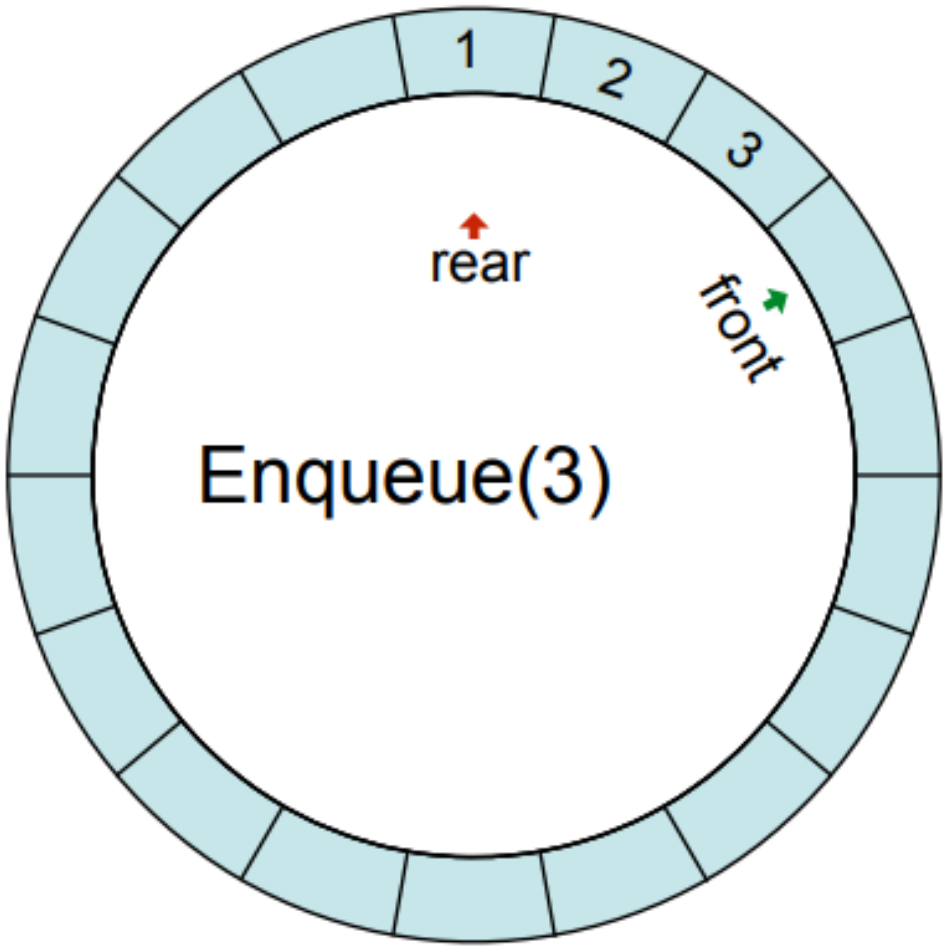
An implementation using a circular buffer.



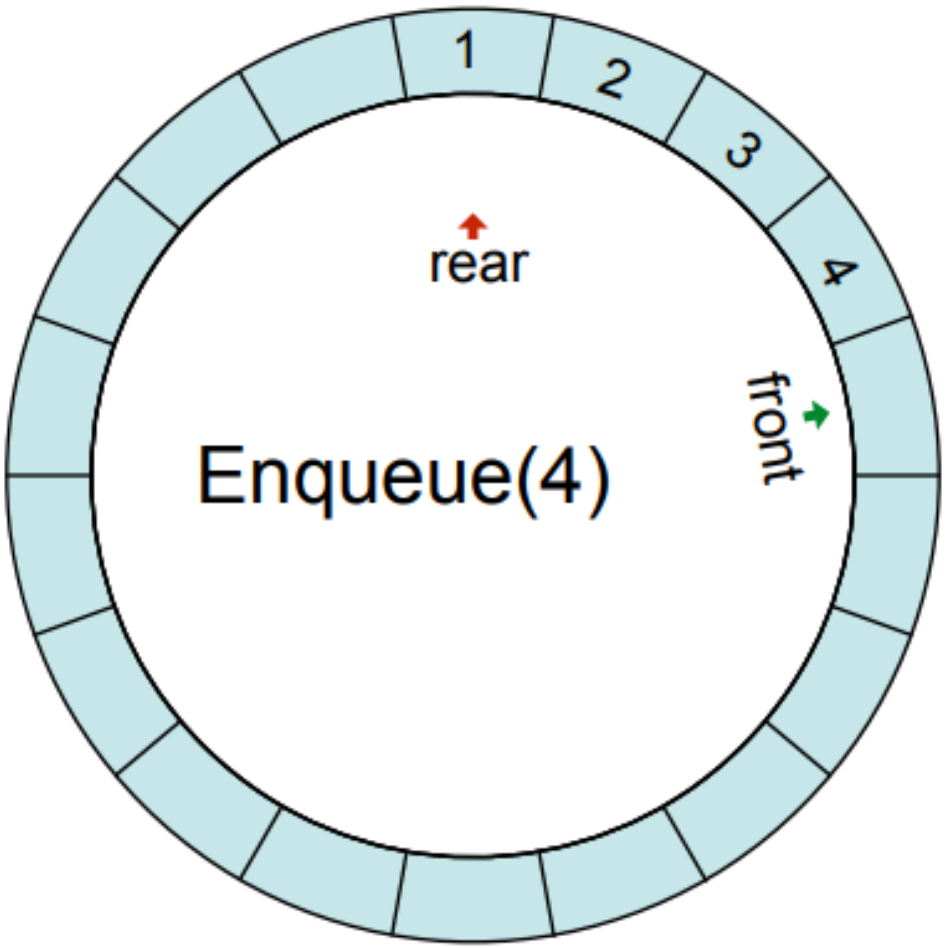
An implementation using a circular buffer.



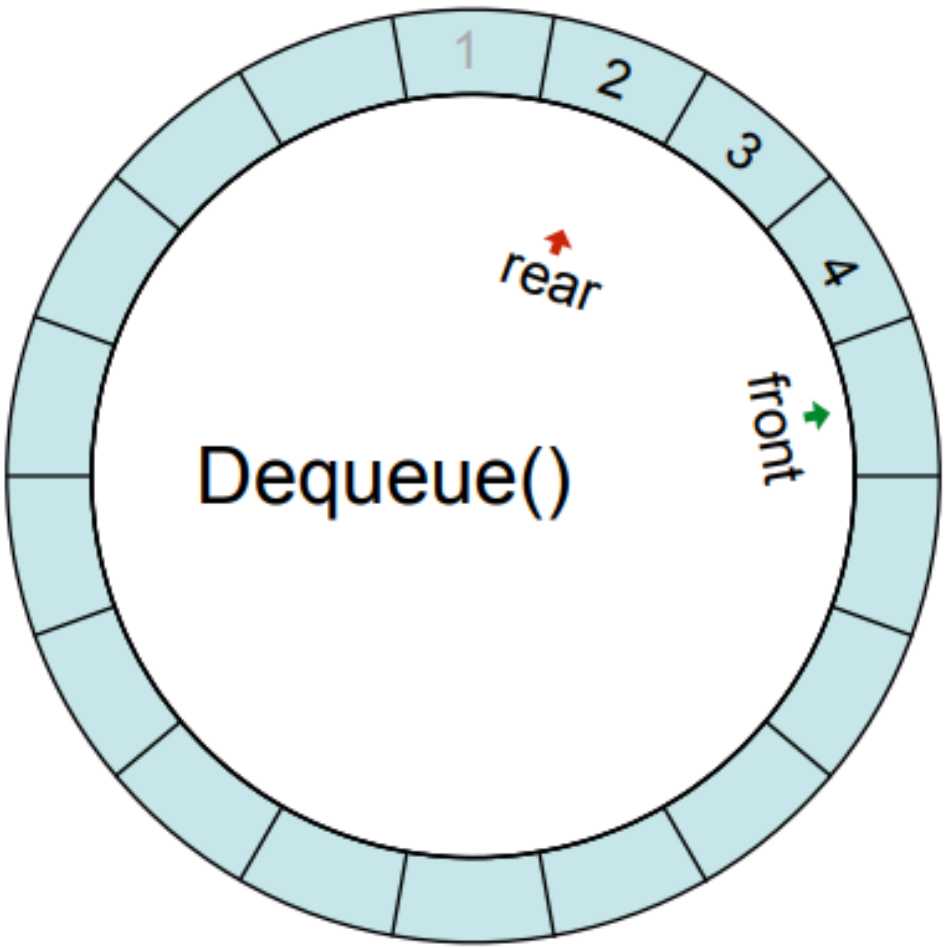
An implementation using a circular buffer.



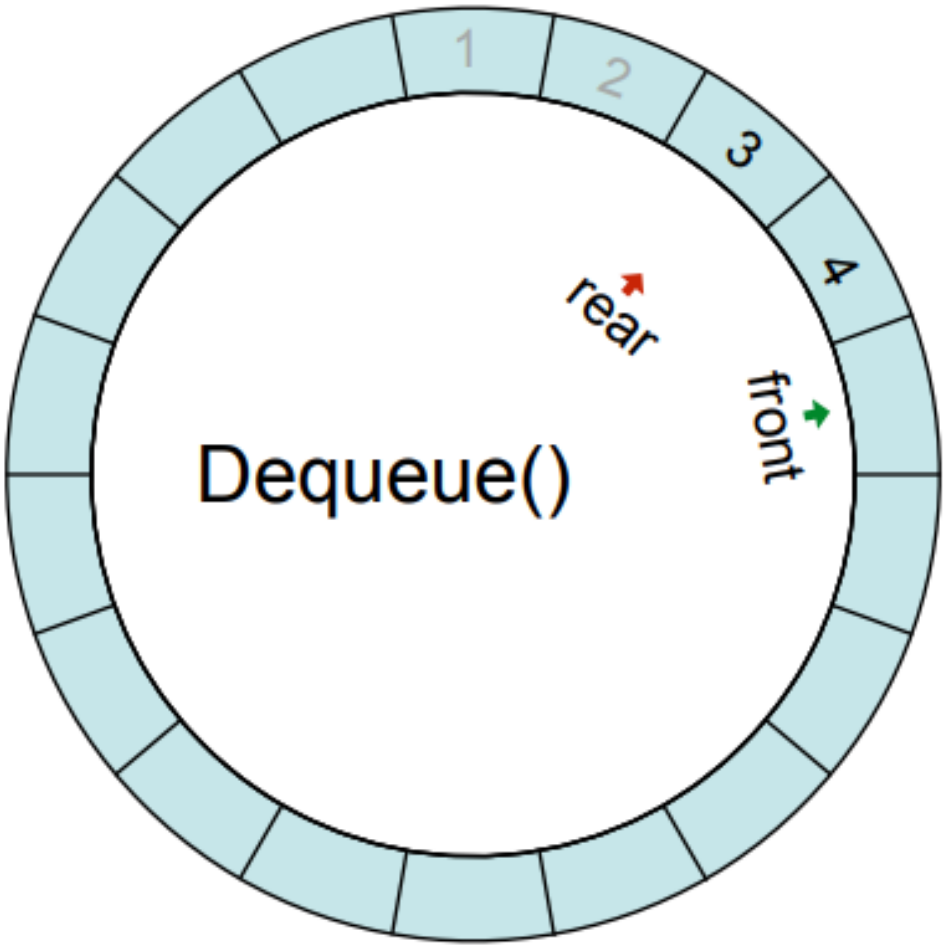
An implementation using a circular buffer.



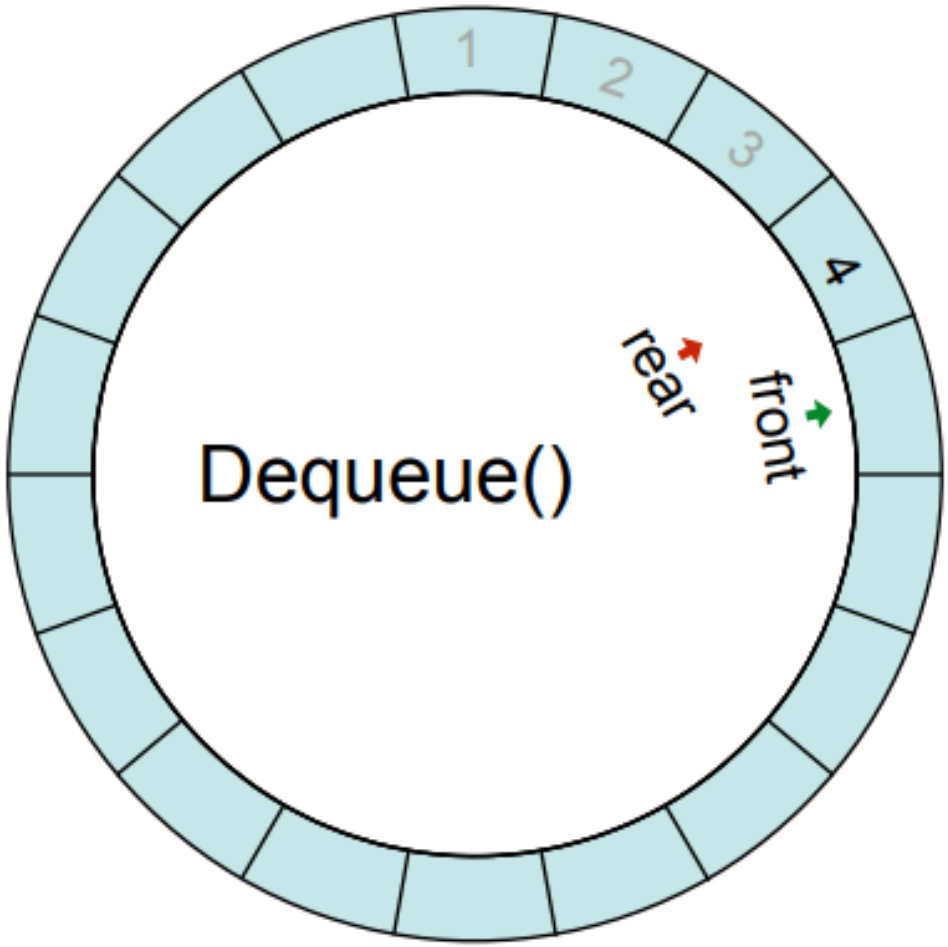
An implementation using a circular buffer.



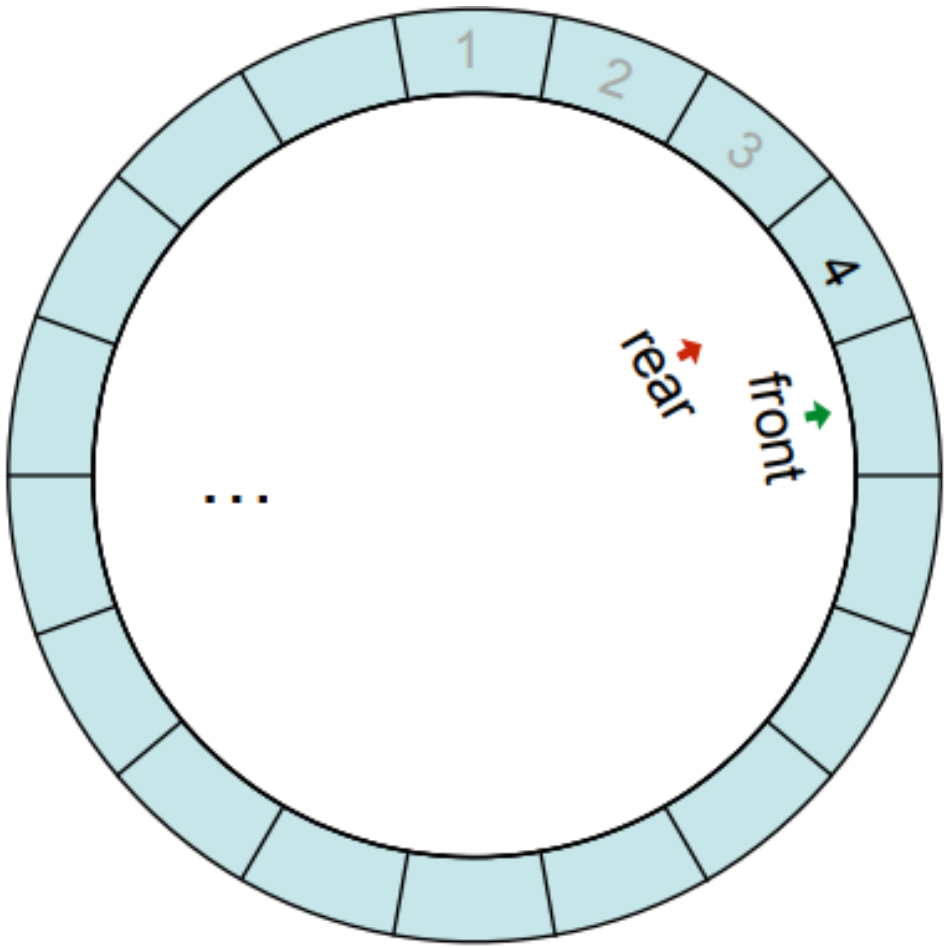
An implementation using a circular buffer.



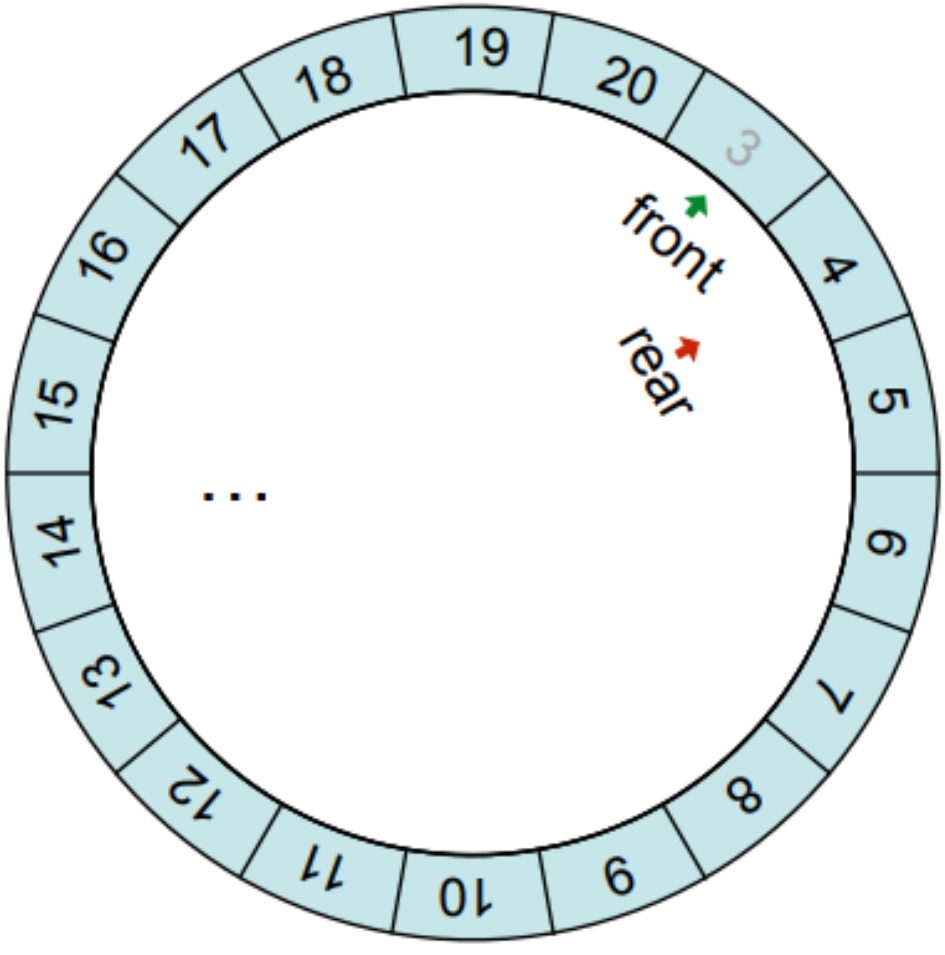
An implementation using a circular buffer.



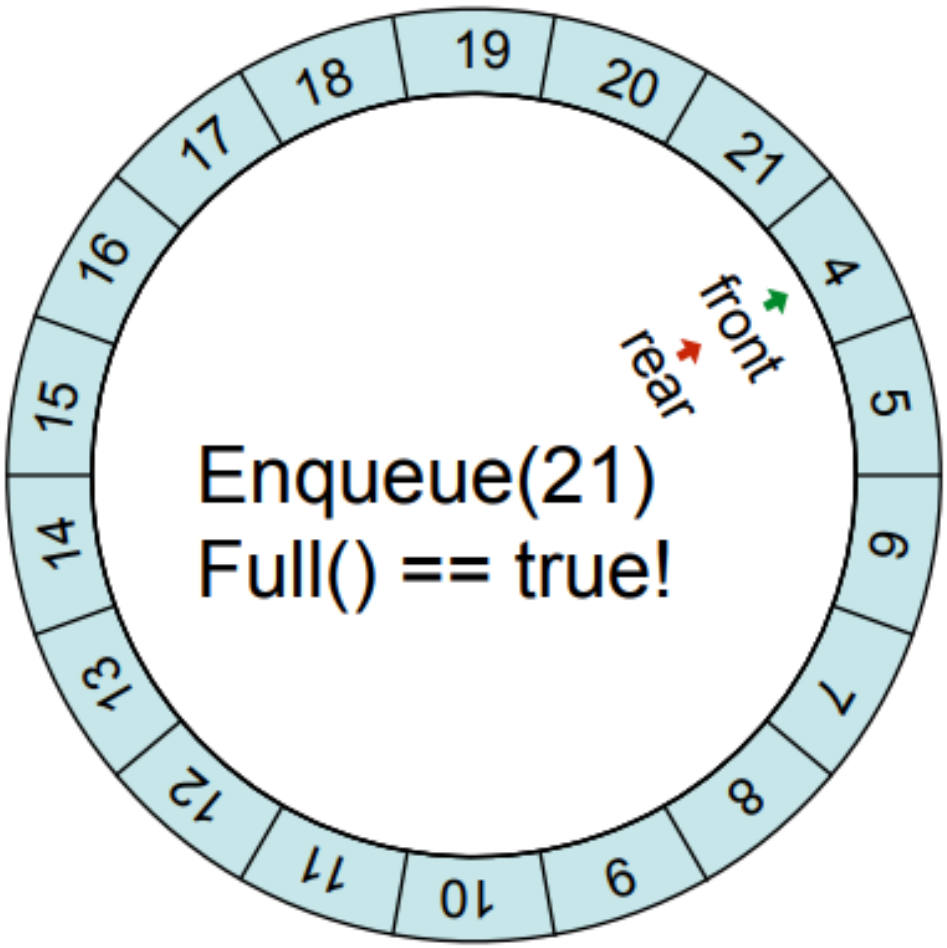
An implementation using a circular buffer.



An implementation using a circular buffer.



An implementation using a circular buffer.



An implementation using a circular buffer.

```
type t = {
  a: array int;
  mutable front: int;
  mutable rear: int;
  mutable number_elts: int;
}

let create (n: int) : t
  requires { 0 < n }
= { a = make n 0;
    front = 0;
    rear = 0;
    number_elts = 0 }
```

A Type Invariant is needed to keep `front` and `rear` within bounds

```
let enqueue (v: int) (t: t) : unit
  (* not able to prove safety *)
= t.a[t.front] <- v;
  t.front <- (t.front + 1) % t.a.length;
  t.number_elts <- t.number_elts + 1

let dequeue (t: t) : int
  (* not able to prove safety *)
= let v = t.a[t.rear] in
  t.rear <- (t.rear + 1) % t.a.length;
  t.number_elts <- t.number_elts - 1;
  v
```

A Type Invariant is needed to keep `front` and `rear` within bounds

```
type t = {
  a: array int;
  mutable front: int;
  mutable rear: int;
  mutable number_elts: int;
} invariant { 0 <= front < a.length }
  invariant { 0 <= rear  < a.length }
```

```
let enqueue (v: int) (t: t) : unit
= t.a[t.front] <- v;
  t.front <- (t.front + 1) % t.a.length;
  t.number_elts <- t.number_elts + 1

let dequeue (t: t) : int
= let v = t.a[t.rear] in
  t.rear <- (t.rear + 1) % t.a.length;
  t.number_elts <- t.number_elts - 1;
  v
```

Safety and Type Invariant post-condition proved.

No need for extra annotations.

Not enough.

- No runtime errors
- but the correct behaviour is not yet ensured
- wrong values may be returned
- valid elements maybe overwritten

```
let main ()
= let q = create 4 in
  enqueue 1 q;
  let ref r = dequeue q in
  r <- dequeue q;
  enqueue 2 q;
  enqueue 3 q; (* ????? *)
  enqueue 4 q;
  enqueue 4 q;
  enqueue 4 q; (* ????? *)
  enqueue 5 q
```

The worst part: the proof succeeds.

The Type Invariant must be refined to ensure that we stay inside the domain of valid queues.

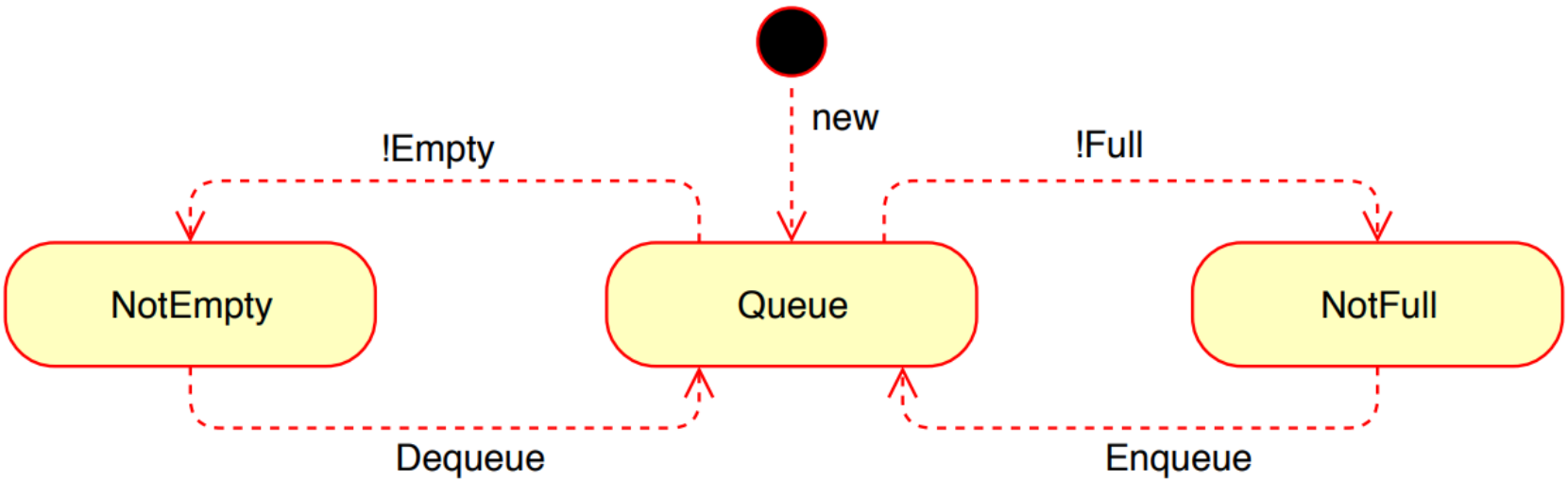
```
type t = {
  a: array int;
  mutable front: int;
  mutable rear: int;
  mutable number_elts: int;
} invariant { 0 <= front < a.length }
  invariant { 0 <= rear  < a.length }
  invariant { if front = rear then
                number_elts = 0 \/
                number_elts = a.length
              else
                number_elts =
                  if front > rear then front - rear
                  else front - rear + a.length }
```

Typestates – Queue

Not there yet.

`Enqueue` and `Dequeue` operations are only valid in certain states.

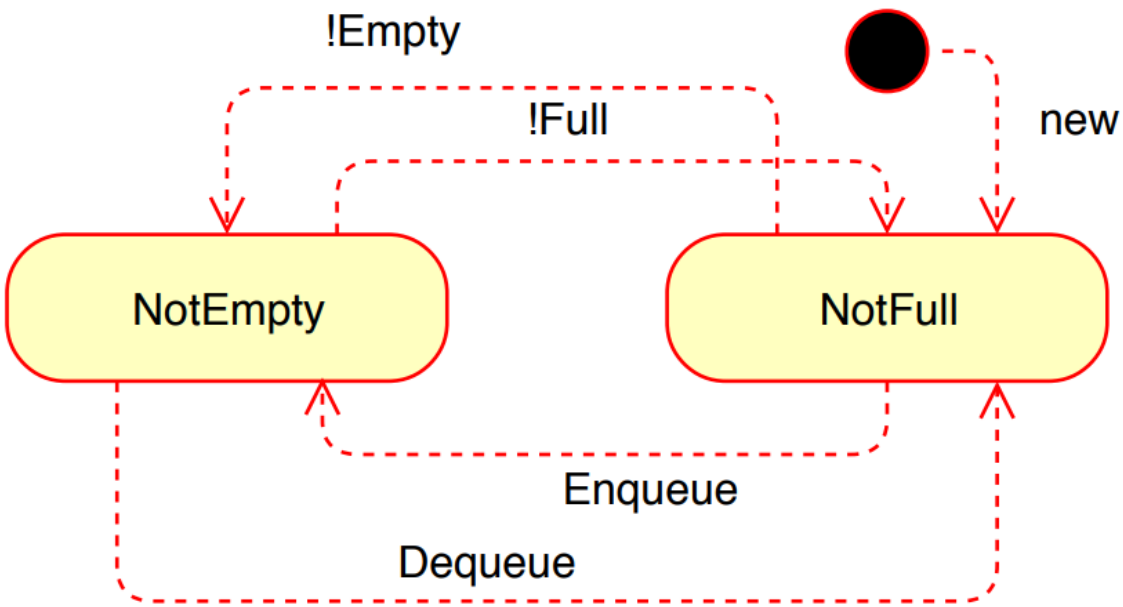
We can obtain such states by dynamic testing operations.



Typestates – Queue

`Enqueue` and `Dequeue` operations are only valid in certain states.

We can obtain such states by dynamic testing operations.



Typestates – Queue

```
predicate not_full (t: t)
= t.number_elts < t.a.length

predicate not_empty (t: t)
= t.number_elts > 0

let create (n: int) : t
  requires { 0 < n }
  ensures  { not_full result }
= ...

let enqueue (v: int) (t: t) : unit
  requires { not_full t }
  ensures  { not_empty t }
= ...

let dequeue (t: t) : int
  requires { not_empty t }
  ensures  { not_full t }
= ...
```

Finally, Dynamic Tests ensure the proper state for a given operation.

```
let is_empty (t: t)
  ensures { result <-> not not_empty t }
= t.number_elts = 0

let is_full (t: t)
  ensures { result <-> not not_full t }
= t.number_elts = t.a.length

let main ()
= let q = create 4 in
  enqueue 1 q;
  let ref r = dequeue q in
  if not is_empty q then begin
    r <- dequeue q;
    enqueue 2 q end;
  if not is_full q then enqueue 3 q;
  if not is_full q then begin
    enqueue 4 q;
    r <- dequeue q end;
  if not is_full q then enqueue 5 q
```