# Designing equals/compareTo/hashCode

The methods should depend

- on fields that are sufficient to identify the entity represented by an object and distinguish it from other entities
- only on fields that will not be changed over time
  - (nice if the fields are declared **final** – so they *can't* be changed.)

If there are no such fields (eg, they might all change over time), or
Every time you create a new object of this type it should be considered unique

Then:

- use the default equals and hashCode which use the object reference/ID/pointer
- compareTo probably doesn't make sense.

# Data Structures and Algorithms

## XMUT-COMP 103 - 2024 T1

## Comparable Objects

Mohammad Nekooei

School of Engineering and Computer Science

Victoria University of Wellington

# Making Classes usable with Collections

- Making a class Comparable makes it easier to use with sort, TreeSet, TreeMap,…

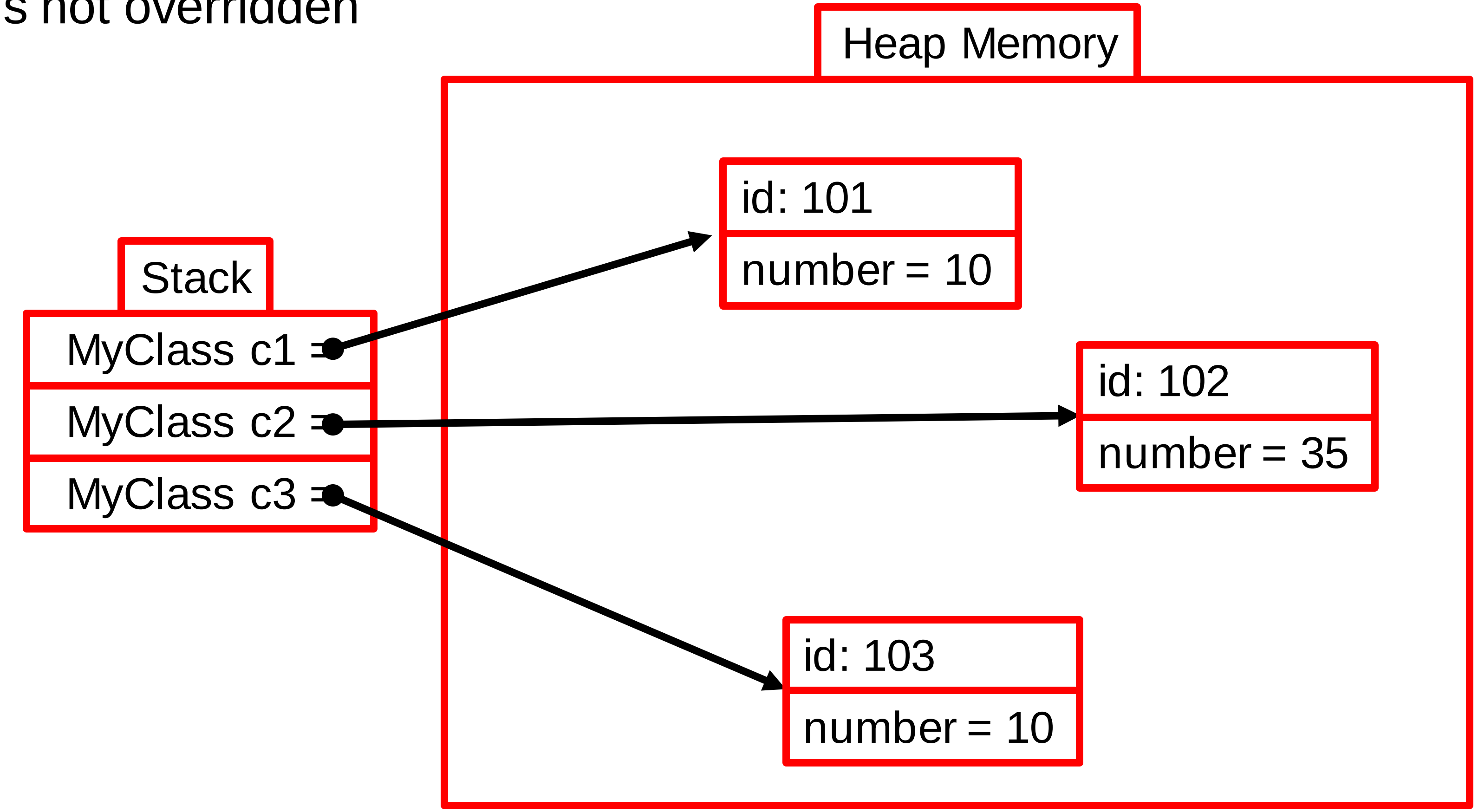
- There are other methods that may also be needed to make your class easy to use:
  - compareTo(…) [to make it Comparable ]
  - toString() [to make it easy to print out objects ]
  - equals(…) [to make everything work,<br>needed if two different objects could be considered to be the same,<br>like Strings]
  - hashCode() [to make HashSet and HashMap work]

- Part of making user-defined Classes usable with Collections
  - Note: compareTo, equals, and hashCode should be consistent!

# Potential problem with equals(…)

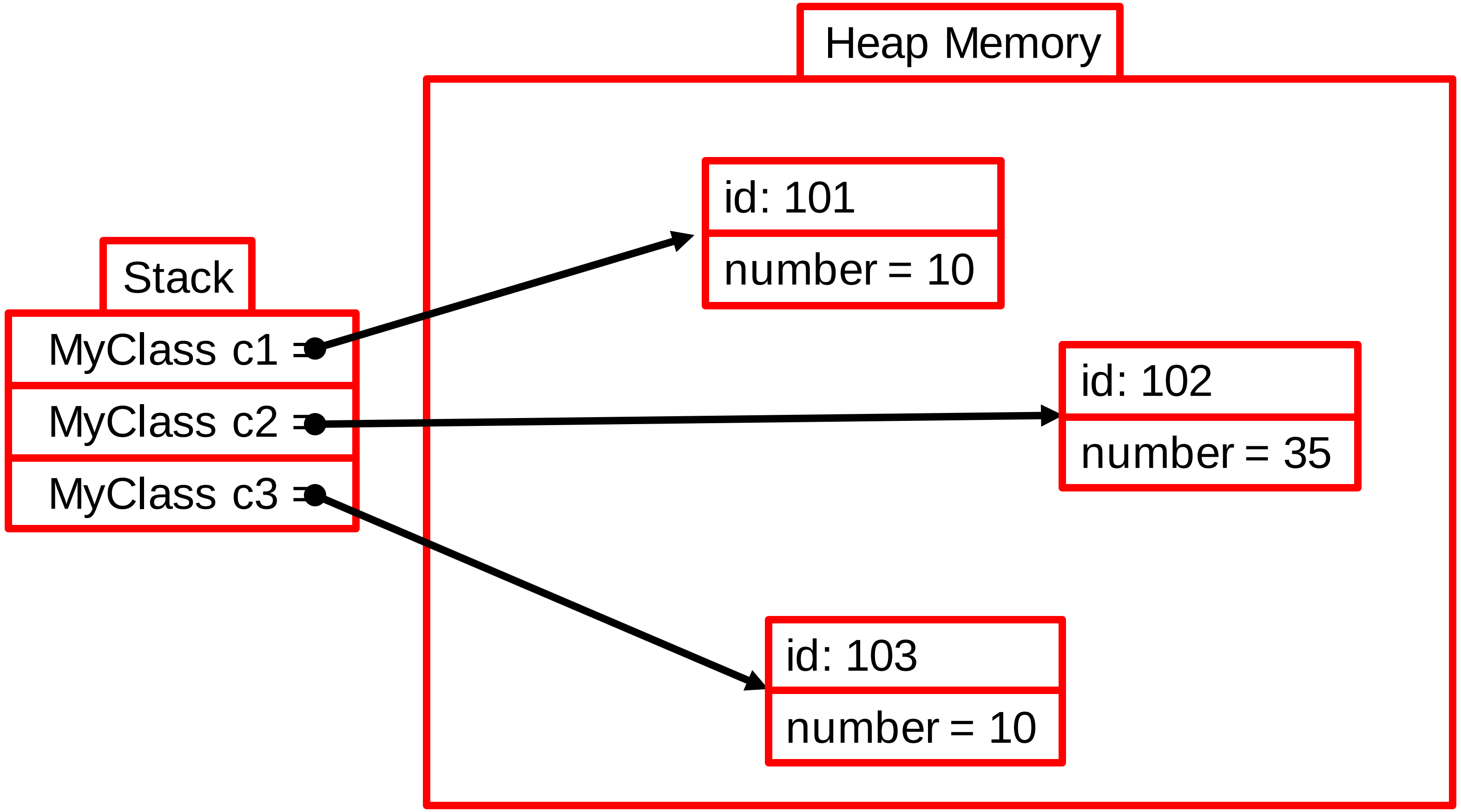
```
public class MyClass {
    private int number;

    //equals is not overridden
}
```



# Potential problem with equals(...)

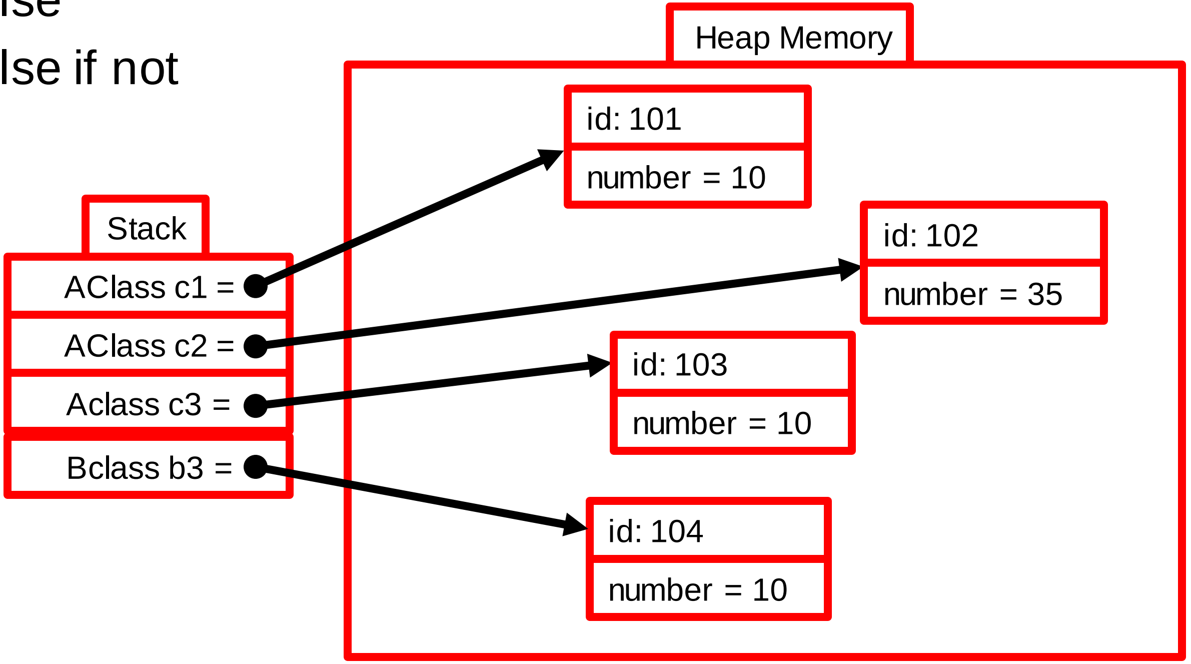
Default equals() checks id of objects. Not the value(s) stored within the object.



# public boolean equals(Object obj){…

- Standard way to override the equals method:
  - Check if obj is the same object as this. → true
  - Check if obj is null → false
  - Check if obj is the same type → false if not
  - Cast obj to this type
  - Check if the fields are the same

```
public class AClass{
   .
   .
   public boolean equals(Object obj){
      if (this==obj)                     { return true; }
      if (obj == null)                   { return false; }
      if (! obj instanceof AClass )      { return false; }
      AClass other = (AClass) obj;
      // check if field values are the same
      return  (this.number==other.number && this.n.equals(other.n) && …) ;
   }
}
```



# public boolean equals(Object obj){…

- Equals for Course class
  - identity based on

```
private String courseCode;                    (but not on lecturer, room, timetable, …)
private int year;
private String trimester;
```

```
public boolean equals(Object obj){
    if (this==obj)                       { return true; }
    if (obj == null)                     { return false; }
    if (! obj instanceof Course )        { return false; }
    Course other = (Course) obj;
    return ( this.courseCode.equals(other.courseCode)  &&
             this.year==other.year  &&
             this.trimester.equals(other.trimester) ) ;
}
```

# Computing Hash Codes

## “Wish list” for a hashCode() method:

- Must produce an integer
- Should take account of all components of the object relevant to identity
- Must be consistent with equals()
  - two items that are equal  must have the same hash value
- Should distribute the hash codes evenly through the range
  - minimises collisions
- Should be fast to compute

# A Simple Hash Function for Strings

- We could add up the ascii codes of all the characters:

```
private int hashCode(String value) {
    int hash = 0;
    for (int i = 0; i < value.length(); i++)
        hash += value.charAt(i);
    return hash;
}
```

Why is this not very good?

# Example: Hashing course codes

418 ← DEAF101

419 ← DEAF102 DEAF201

⋮

429 ← BBSC201 MDIA101

430 ← ECHI410 MDIA102 MDIA201

431 ← ECHI303 JAPA111 JAPA201 MDIA202 MDIA220 MDIA301

432 ← ARCH101 ASIA101 BBSC231 BBSC303 BBSC321 CHEM201 ECHI403 ECHI412 JAPA112 JAPA211 JAPA301 MDIA203 MDIA302 MDIA320

⋮

450 ← ANTH412 ARCH389 ARTH111 BIOL228 BIOL327 BIOL372 CHEM489 COML304 COML403 COML421 COMP102 COMP201 CRIM313 CRIM421 DESN215 DESN233 ECON328 ECON409 ECON418 ECON508 EDUC449 EDUC458 EDUC548 EDUC557 ENGL228 ENGL408 ENGL426 ENGL435 ENGL444 ENGL453 FREN124 FREN331 FREN403 FREN412 GEOL362 GEOL407 GERM214 GERM403 GERM412 INFO213 INFO312 INFO402 ITAL206 ITAL215 LALS501 LATI404 LING224 LING323 LING404 MAOR102 MARK304 MARK403 MATH206 MATH314 MATH323 MATH431 MOFI403 PHIL104 PHIL203 PHIL302 PHIL320 PHIL401 PHIL410 RELI321 RELI411 SAMO101

⋮

# Better Hash Functions

- Make the contribution of each component depend on its position:

```
public class CourseOffering{
    private final String courseCode;
    private final int year;
    private final char trimester;
    private …    // other fields for timetable, coordinator, …..

    /** hash code depends on the course code, the year, and the trimester */
    public int hashCode() {
        int prime = 104417;
        int hash = year;
        for (int i = 0; i < courseCode.length(); i++)
            hash = hash * prime + courseCode.charAt(i);
        hash = hash * prime  + trimester;
        return hash;
    }

 …
```

# Data Structures and Algorithms

## XMUT-COMP 103 - 2024 T1

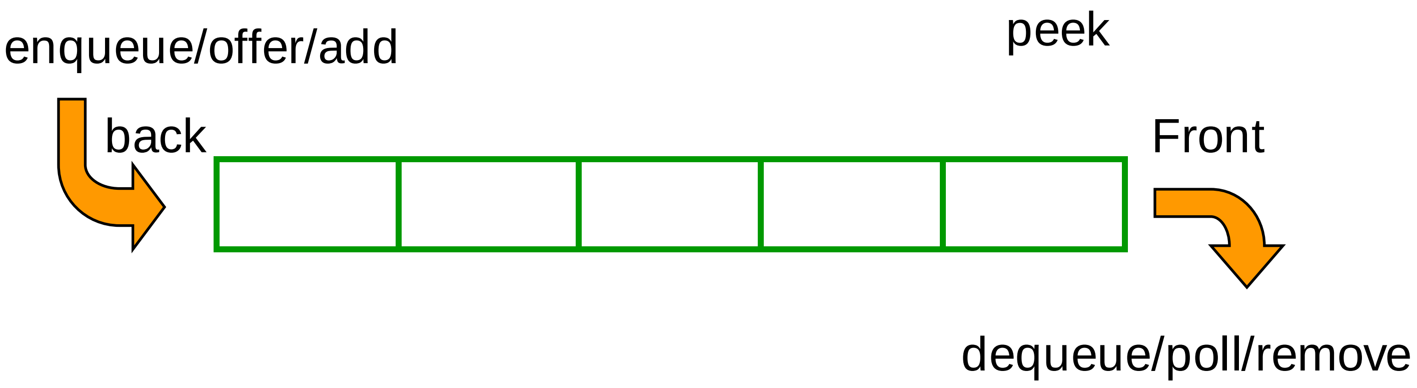
## Queues

Mohammad Nekooei

School of Engineering and Computer Science

Victoria University of Wellington

# Queues



- Collection of items in order
  - like Lists and Stacks
- Main operations:
  - enqueue: put item on the queue
  - dequeue: remove item from front of the queue
- These operations should be efficient.
  - Shouldn't get much more expensive if the queue is very large
- A Queue is a Collection:
  - THEREFORE: other operations – contains(…), remove(…), etc – also work

    BUT, they not be efficient.

# Queue operations

- isEmpty(),
- size(),
- clear()

- offer(E item)    enqueue
- add(E item)    enqueue

- poll() → E    dequeue    (returns null if queue is empty)
- remove() → E    dequeue    (throws exception if queue is empty)

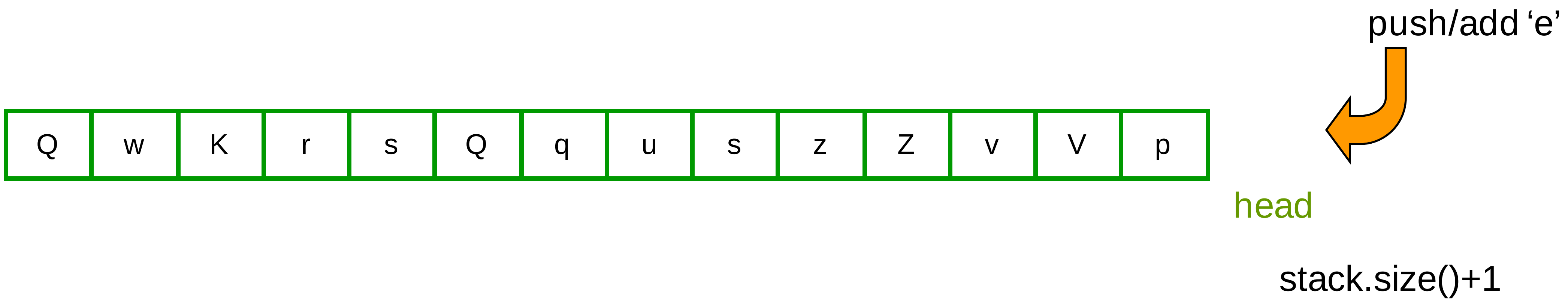
- peek() → E    look at front    (returns null if queue is empty)
- element() → E    look at front    (throws exception if queue empty)

# Queues and efficiency

- The main operators of queues should be efficient.
    - the time it take to do them should be fast
    - especially important when they grow in size <= a constant speed is needed!
- Let’s investigate how stacks can be implemented efficiently.

# Stacks and efficiency

- You can use an ArrayList to implement a Stack (LIFO) efficiently:



push/add ‘e’

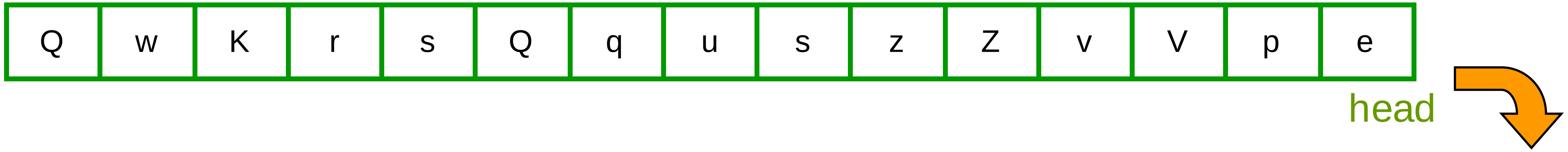
| | | | | | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| Q | w | K | r | s | Q | q | u | s | z | Z | v | V | p |

head

stack.size()+1

# Stacks and efficiency

- You can use an ArrayList to implement a Stack (LIFO) efficiently:



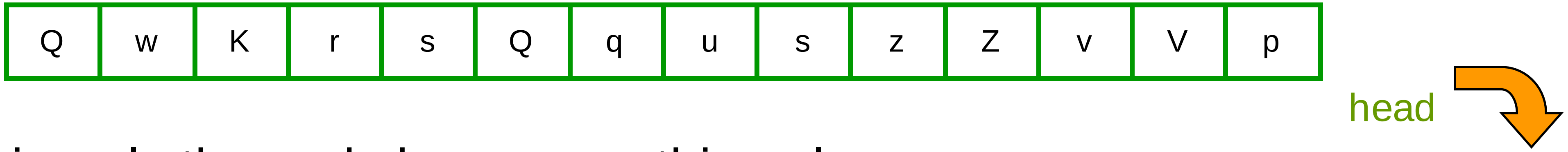
| | | | | | | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| Q | w | K | r | s | Q | q | u | s | z | Z | v | V | p | e |

head

- No changes to the stack other than an ’e’ was added at the end.

# Stacks and efficiency

- You can use an ArrayList to implement a Stack (LIFO) efficiently:



| | | | | | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| Q | w | K | r | s | Q | q | u | s | z | Z | v | V | p |

pop/remove
returns ‘e’
stack size -1

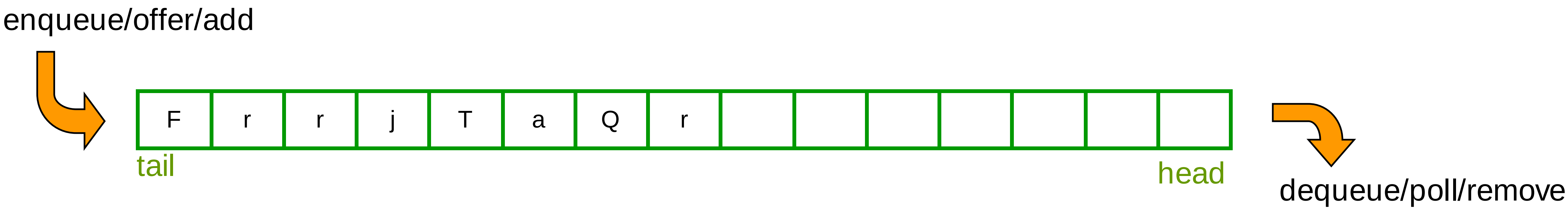
- Again only the end changes, nothing else
- push and pop at the end➔ O(1)

- Stacks are naturally efficient with an ArrayList!

# Queues and efficiency

- What about a Queue (FIFO) ?

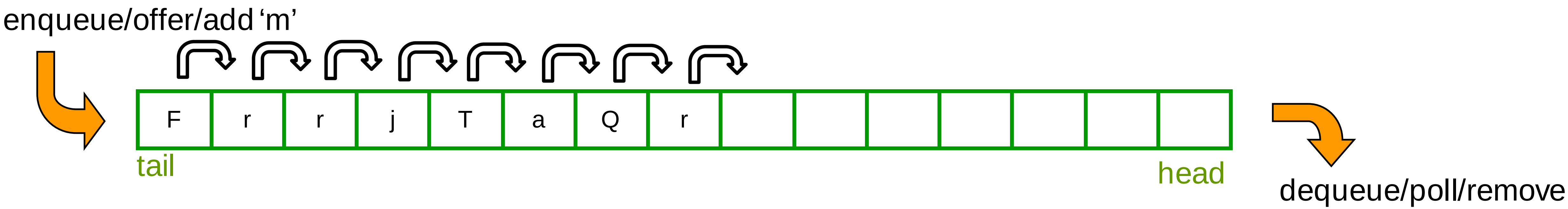
- Dequeue works like a stack, so is fast ➔ O(1)



# Queues and efficiency

- What about a Queue (FIFO) ?

- Dequeue works like a stack, so is fast ➔ O(1)
- Enqueue requires shifting every item up one place to add
  - It “costs” the current length (n) to move ➔ O(n)



enqueue/offer/add ‘m’

| | | | | | | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| F | r | r | j | T | a | Q | r | | | | | | | |

tail

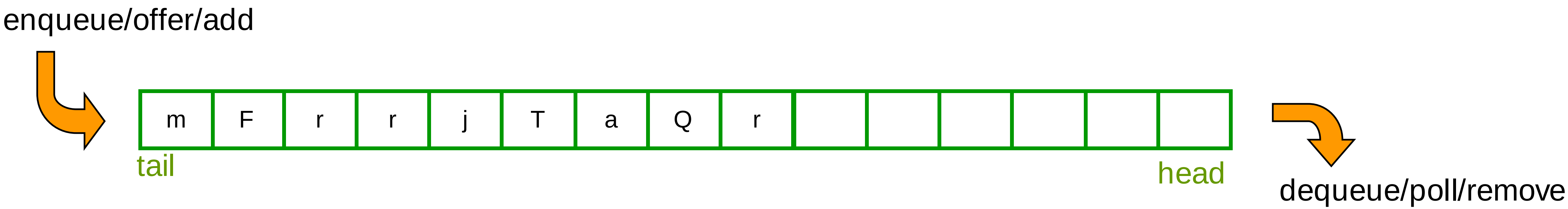
head

dequeue/poll/remove

# Queues and efficiency

- What about a Queue (FIFO) ?

- Dequeue works like a stack, so is fast ➔ O(1)
- Enqueue requires shifting every item up one place to add
  - It “costs” the current length (n) to move ➔ O(n)



enqueue/offer/add

| m | F | r | r | j | T | a | Q | r |  |  |  |  |  |  |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|

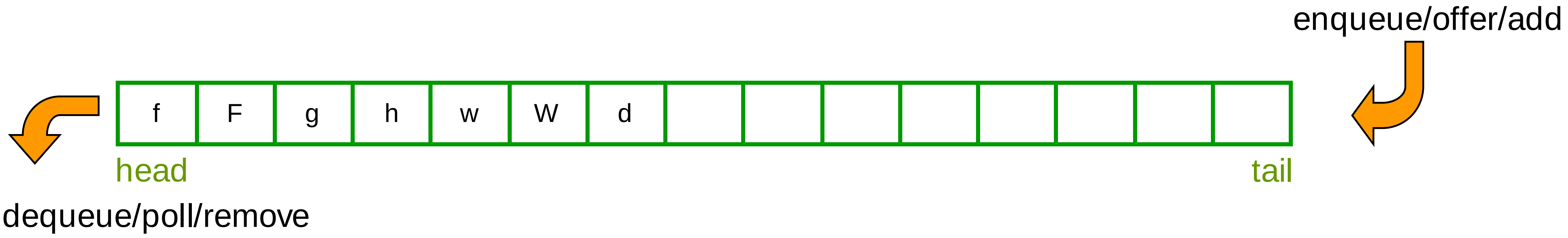
tail

head

dequeue/poll/remove

# Queues and efficiency

- What about a Queue, The other way round?



enqueue/offer/add

|   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| f | F | g | h | w | W | d |   |   |   |   |   |   |   |   |

head

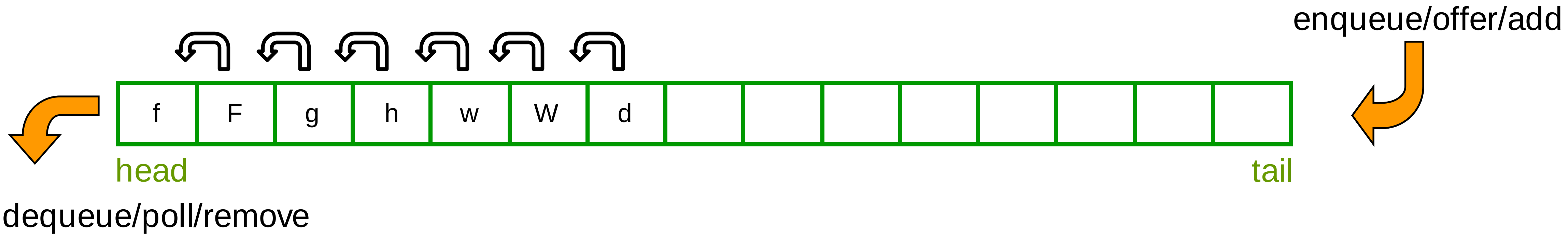
tail

dequeue/poll/remove

- Enqueue is like push on a stack, so it is fast ➔ O(1)

# Queues and efficiency

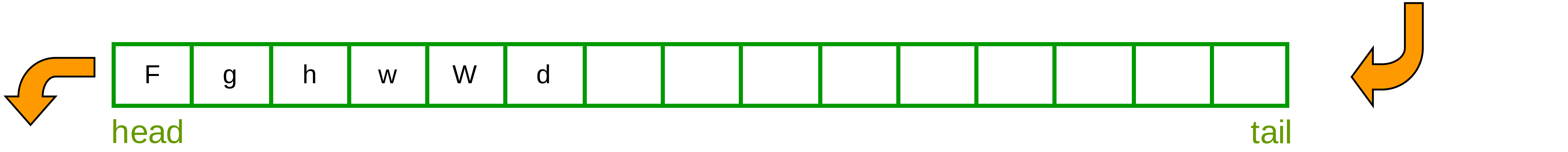
- What about a Queue, The other way round?



- Enqueue is like push on a stack, so it is fast → O(1)
- Dequeue requires shifting every item down one place → O(n)

# Queues and efficiency

- What about a Queue, The other way round?



dequeue/poll/remove

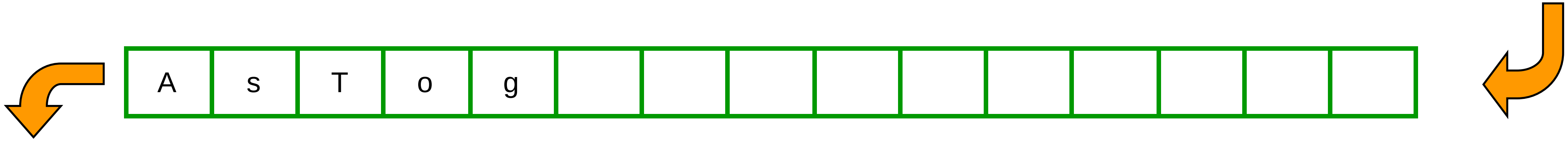
- Enqueue is like push on a stack, so it is fast ➔ O(1)
- Dequeue requires shifting every item down one place ➔ O(n)

- Big Oh notation:
  - O(1) : fixed number of steps, regardless of how big the collection is
  - O(n) : number of steps proportional to the size of the collection.

# Queues and efficiency

- Using an array and two indexes:

enqueue/offer/add



| | | | | | | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| A | s | T | o | g | | | | | | | | | | |

head

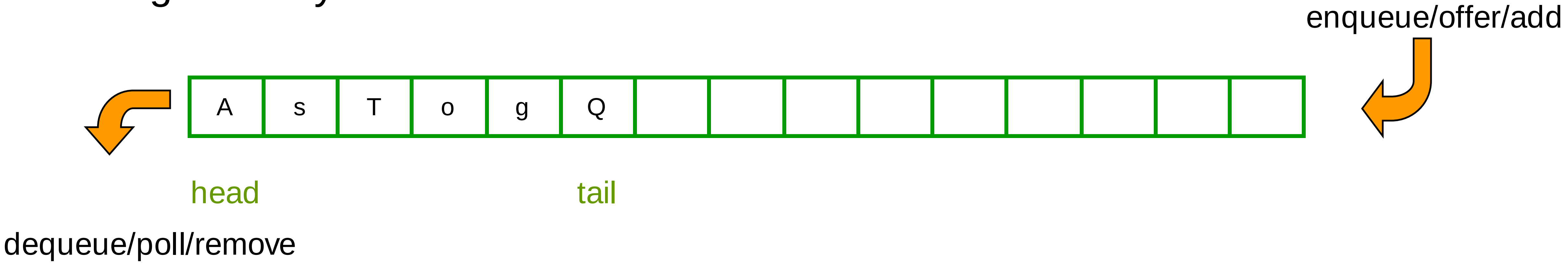
tail

dequeue/poll/remove

- Enqueue:
  - Get tail
  - tail++
  - Add new value at new tail

# Queues and efficiency

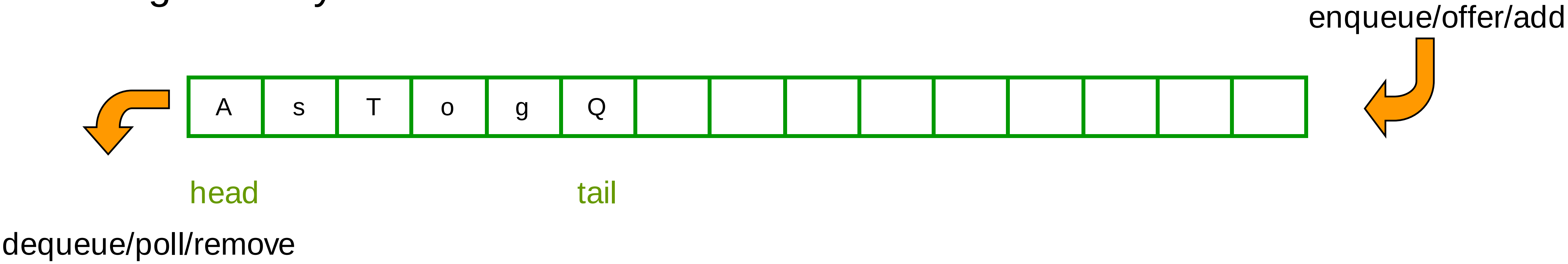
- Using an array and two indexes:



- Enqueue:
  - Get tail
  - tail++
  - Add new value at new tail

# Queues and efficiency

- Using an array and two indexes:



enqueue/offer/add

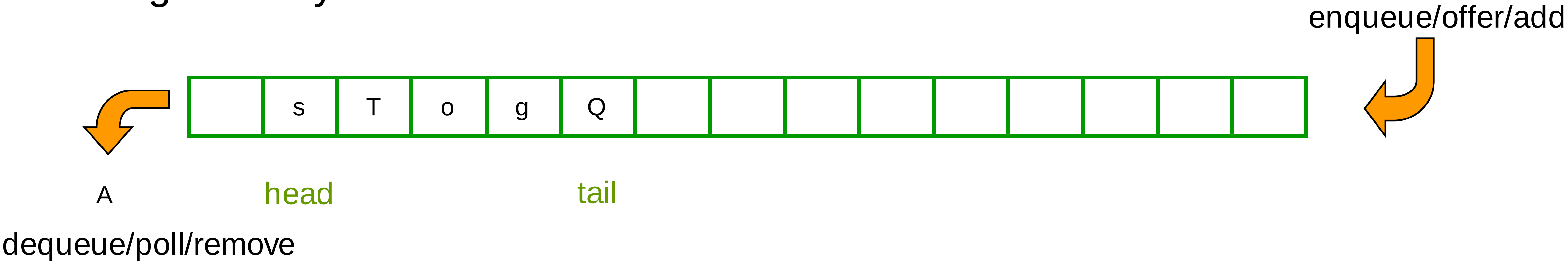
| | | | | | | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| A | s | T | o | g | Q | | | | | | | | | |

dequeue/poll/remove

- Enqueue is fast ➔ O(1)
- Dequeue:
  - Return value at head
  - head++

# Queues and efficiency

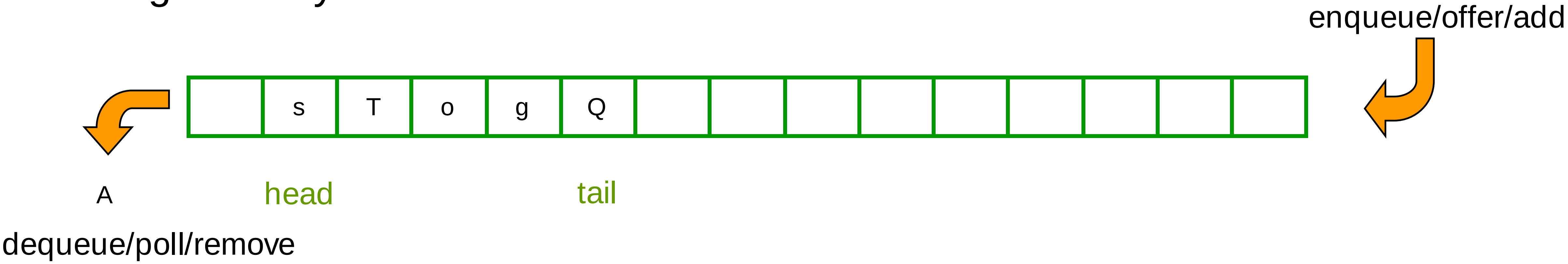
- Using an array and two indexes:



- Enqueue is fast ➔ O(1)
- Dequeue:
  - Return value at head
  - head++

# Queues and efficiency

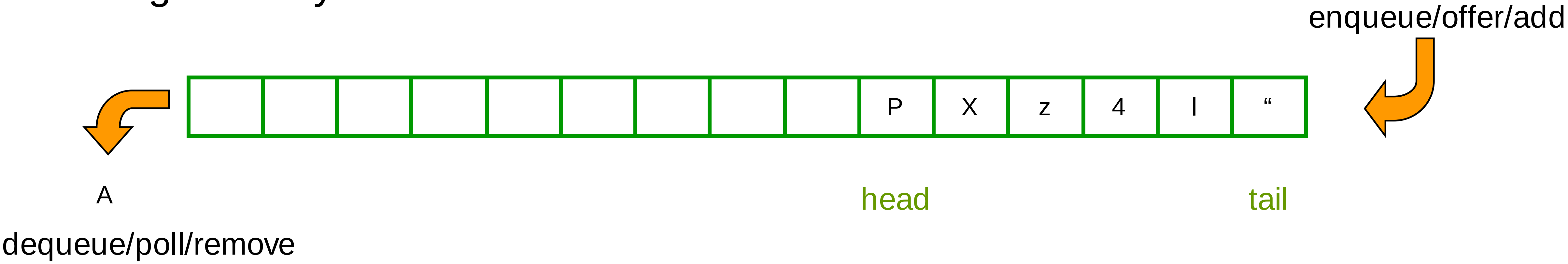
- Using an array and two indexes:



- Enqueue is fast ➔ O(1)
- Dequeue is fast ➔ O(1)

# Queues and efficiency

- Using an array and two indexes:

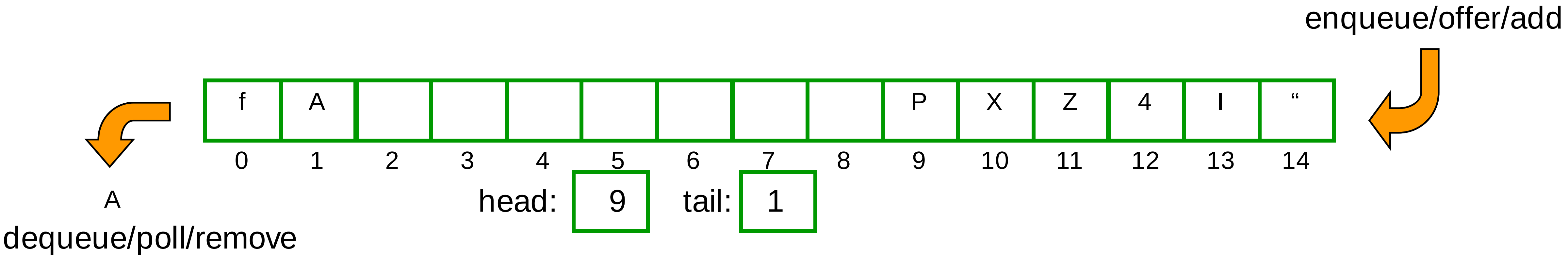


- Enqueue is fast ➔ O(1)
- Dequeue is fast ➔ O(1)

- What about space? (memory)

# Queues and efficiency

- Using an array and two indexes:



enqueue/offer/add

| 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 | 11 | 12 | 13 | 14 |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| f | A | | | | | | | | P | X | Z | 4 | I | “ |

head: 9 tail: 1

A

dequeue/poll/remove

- Enqueue is fast ➔ O(1)
- Dequeue is fast ➔ O(1)

- What about space? (memory)
- “Wrap around” at the end;

# Java Implementations

- Java classes for Queue:
  - ArrayDeque `Queue<Patient> waitingRoom = new ArrayDeque<Patient>();`
  - LinkedList

- ArrayDeque is actually a kind of Deque – an extension of Queue:
  - Deque = Double Ended Queue
  - Add or remove at either end.
  - Includes Stacks and Queue

  - offer(e) = offerLast(e)
  - push(e) = offerFirst(e)
  - poll() = pop() = pollFirst()
  - - = pollLast()
  - peek() = peekFirst()
  - - = peekLast()

# Data Structures and Algorithms

## XMUT-COMP 103 - 2024 T1

## Priority Queues

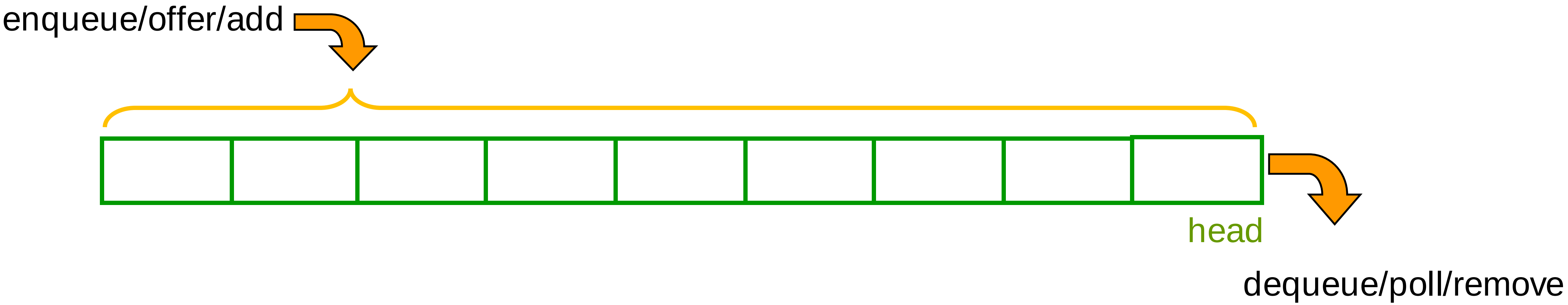
**Mohammad Nekooei**

**School of Engineering and Computer Science**

**Victoria University of Wellington**

# Priority Queues

- Priority Queues
  - Items ordered by priority, instead of arrival order
    - dequeue/poll → highest priority item (earliest in the ordering



# Priority Queues: ordering

Ordering:

- Highest priority = earliest in ordering.
- Typically high priority = 1, low priority = 10 (large number)

Specify ordering like with Collections.sort():

either

- use natural ordering of the items using compareTo (if they are Comparable)

```
Queue<Patient> waitingRoom = new PriorityQueue<Patient>();
```

or

- give the Priority Queue a compare(…) function when created:

```
Queue<Patient> waitingRoom =
    new PriorityQueue<Patient>((Patient p1, Patient p2) ->{
                                    if (p1.getPri()>p2.getPri()){ return -1:} else if (p1 …   }  );
```

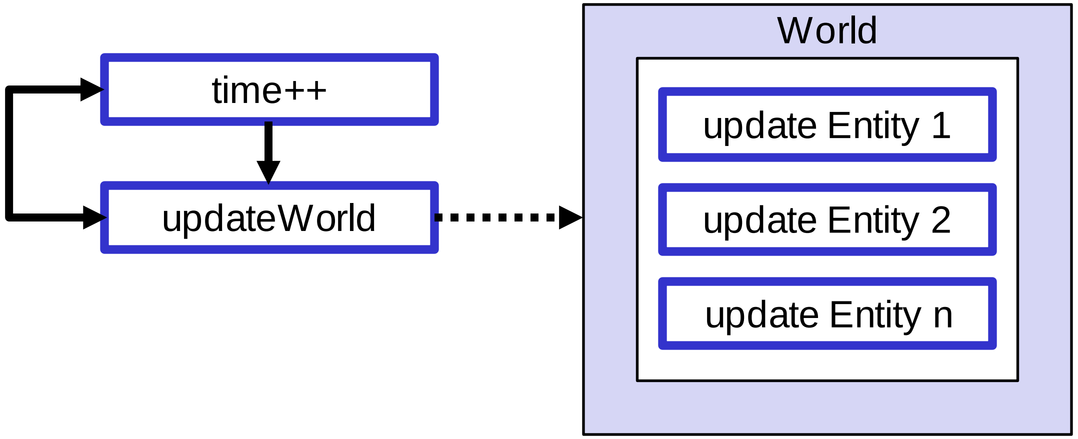
# Applications of Queues and PriorityQueues

Many applications! (and many specialised Queue classes)

- Operating Systems, Network Applications, Multi-User Systems
  - Handling requests/events/jobs that must be done in order
  - (often called a “buffer” in this context)
- Simulations
  - Representing queues in the real world (traffic, customers, deliveries, ….)
  - Managing the events that must happen in the future
- Programs to control delivery of orders or manage customers/clients
- Search Algorithms
  - breadth-first search
  - breadth-first search

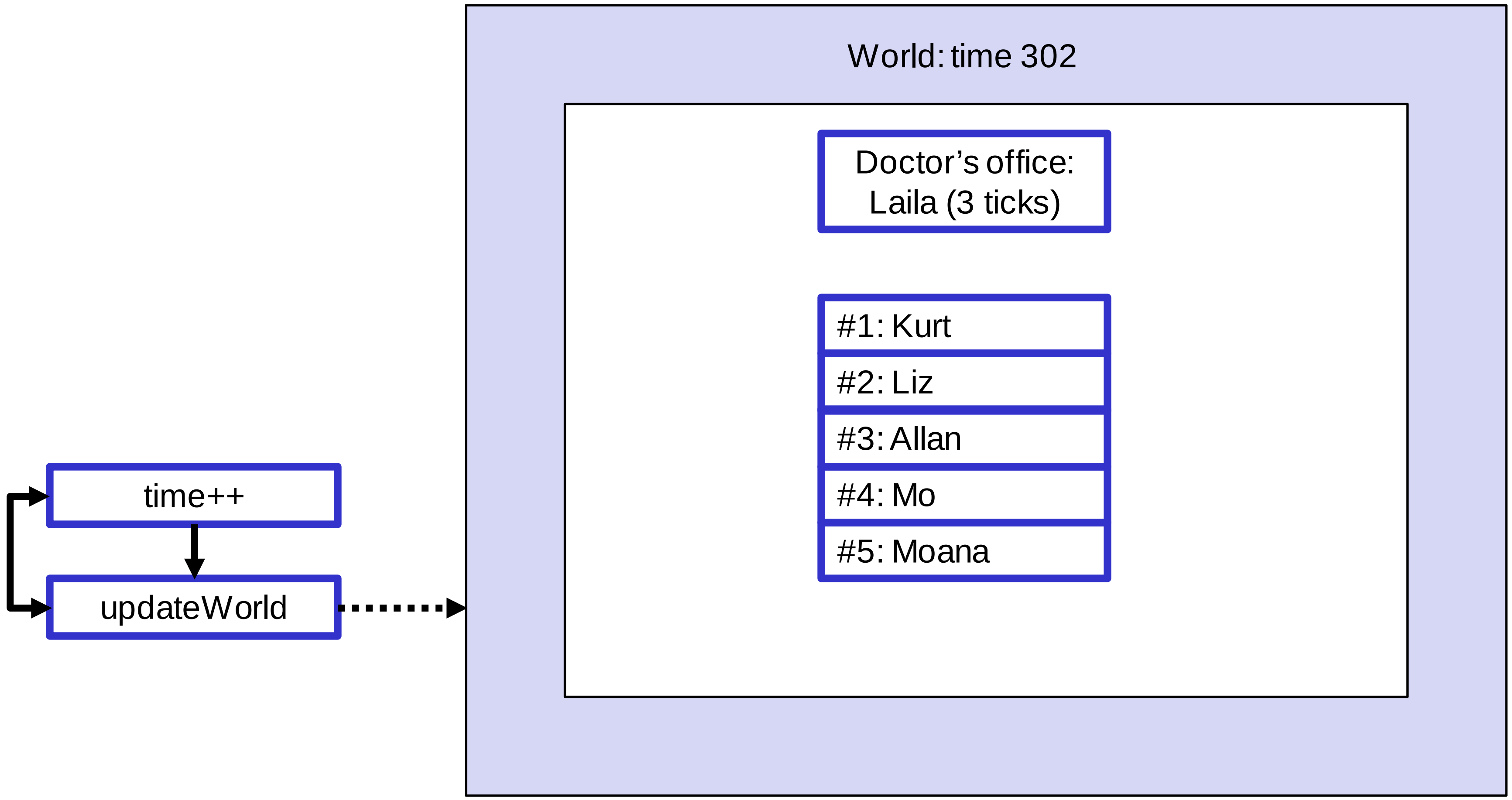
# Simulation

- Tick-based simulation
  - Time is discrete
  - Main loop advances time by one tick for each iteration
  - Each tick, update the state of every entity in the world by one tick
  - Efficient if every entity changes every tick;
    May be inefficient if not very much happens most of the time
  - Often used in games



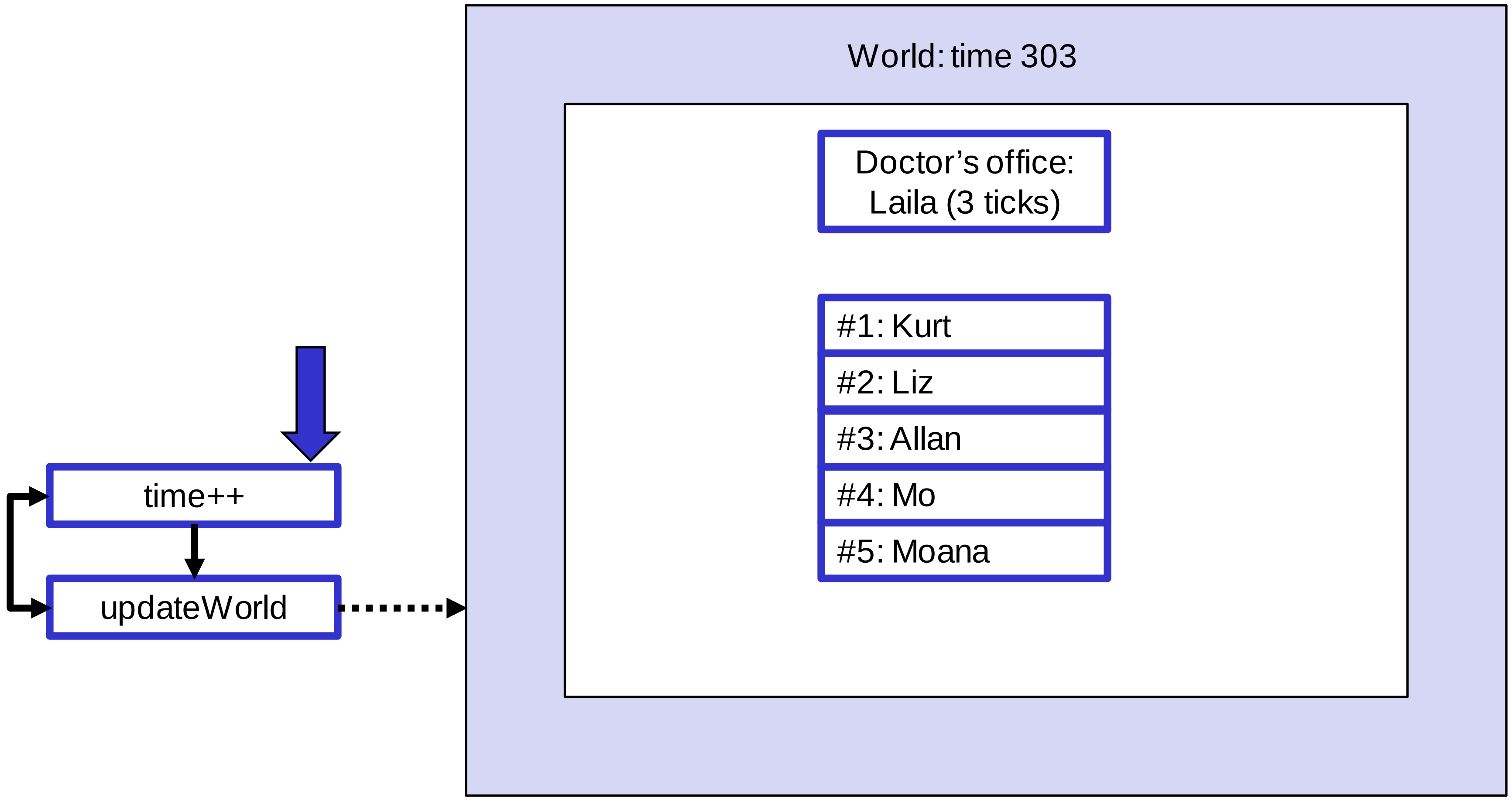
# Simulation of Queues

- Doctor’s waiting room



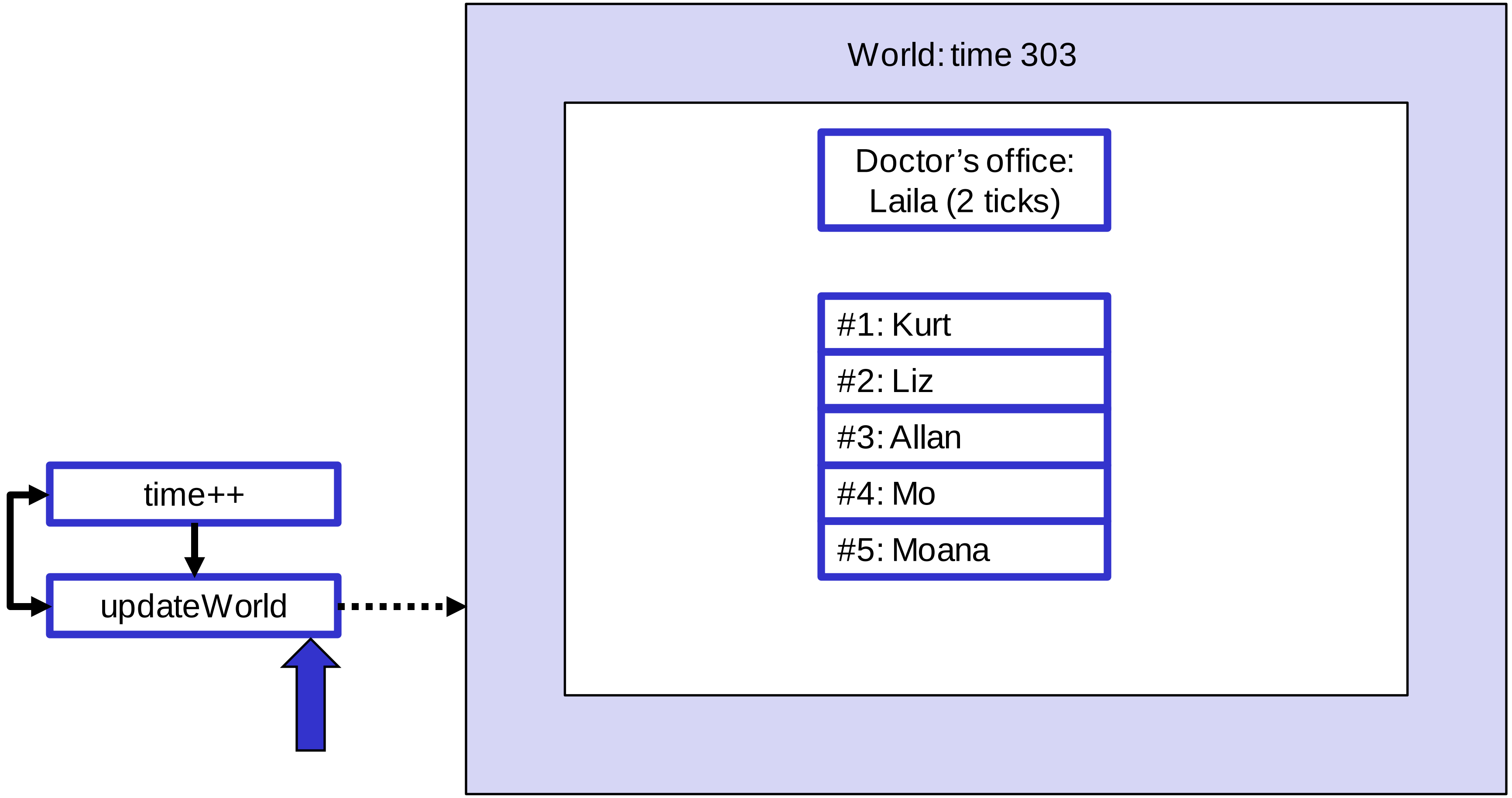
# Simulation of Queues

- Doctor’s waiting room



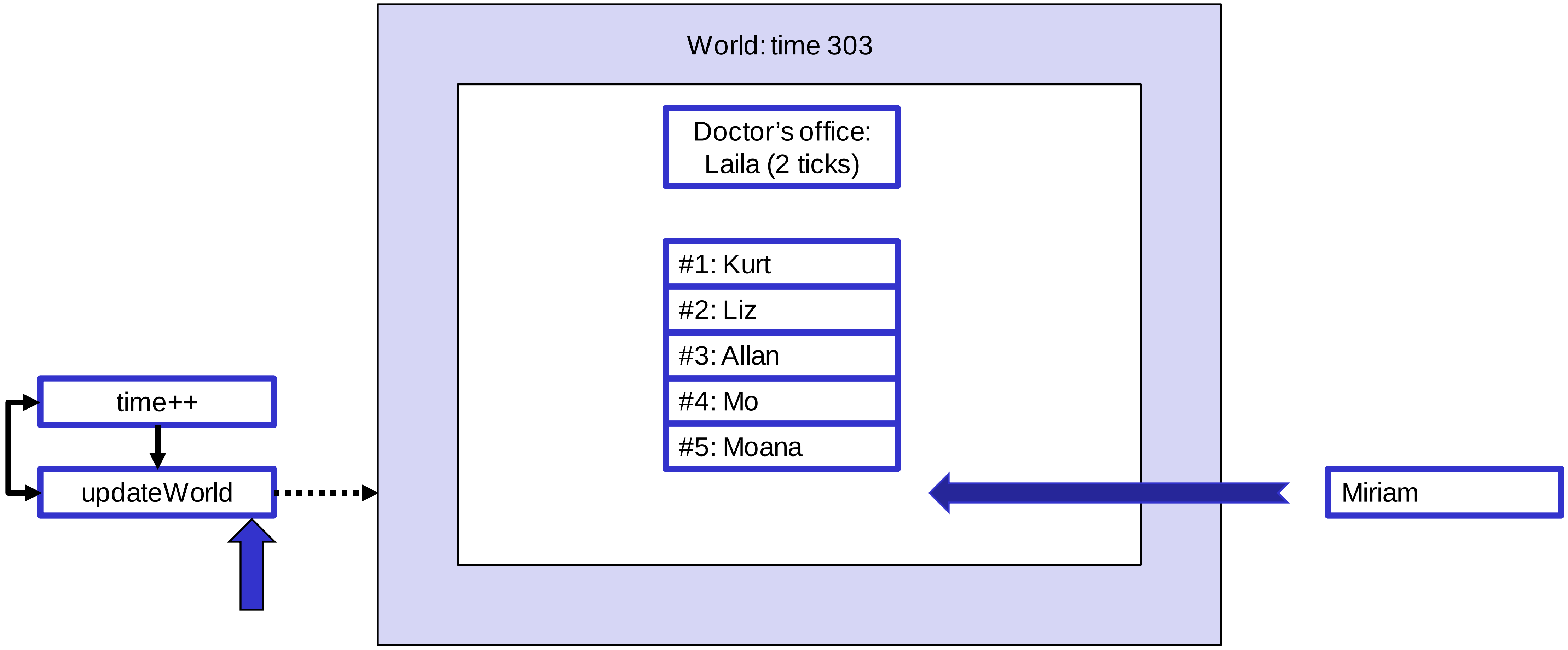
# Simulation of Queues

- Doctor’s waiting room



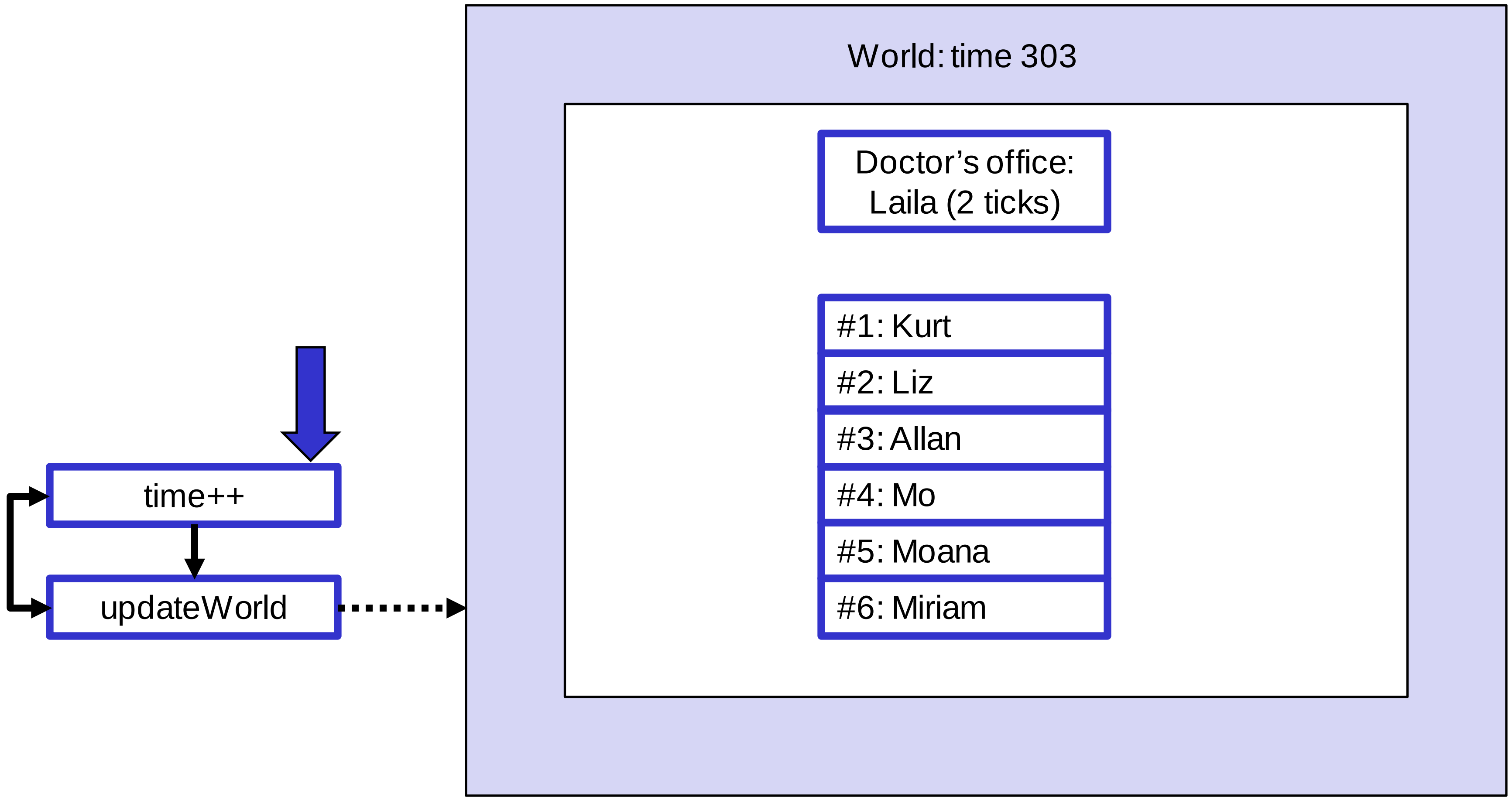
# Simulation of Queues

- Doctor’s waiting room



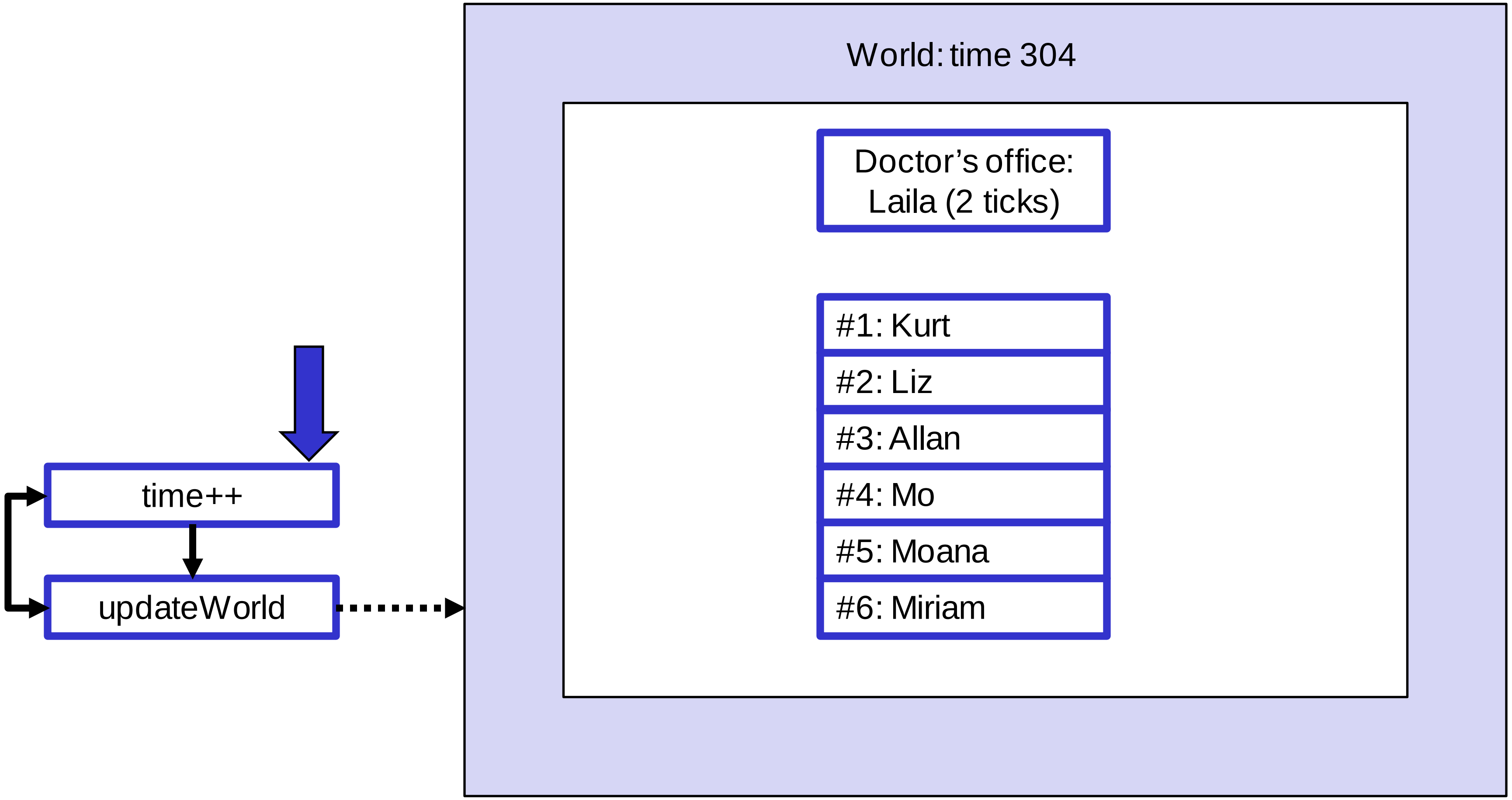
# Simulation of Queues

- Doctor’s waiting room



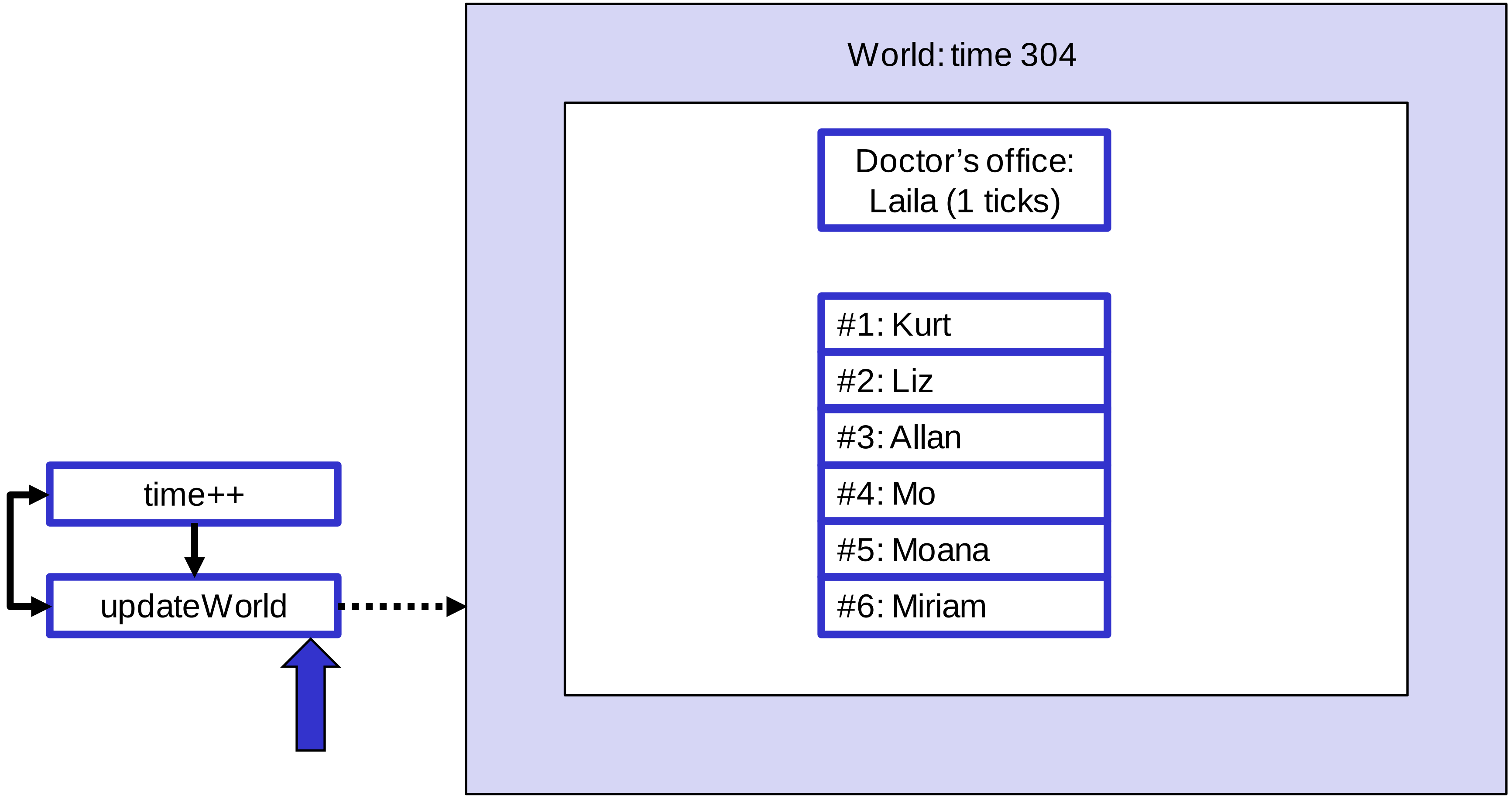
# Simulation of Queues

- Doctor’s waiting room



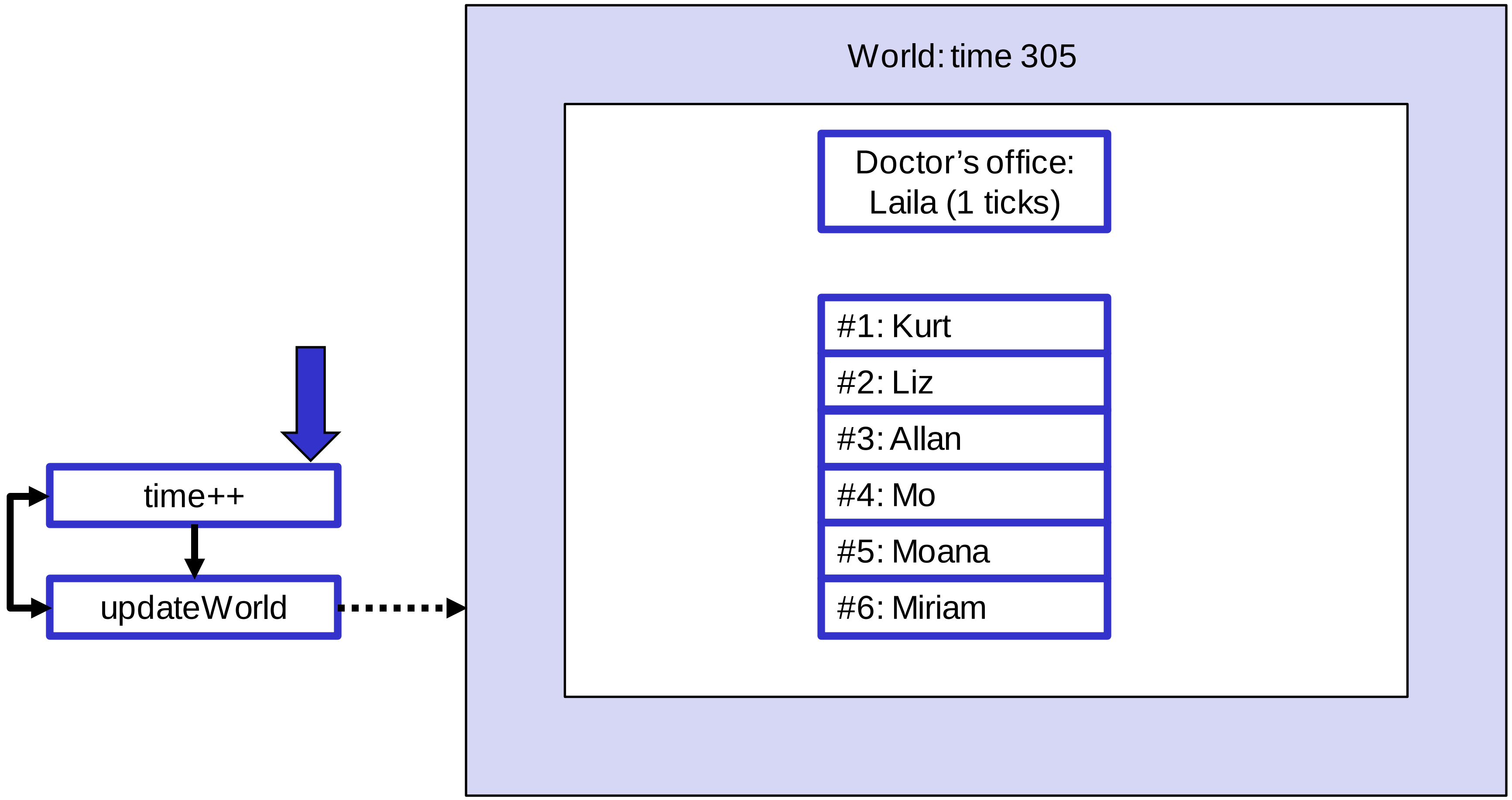
# Simulation of Queues

- Doctor’s waiting room



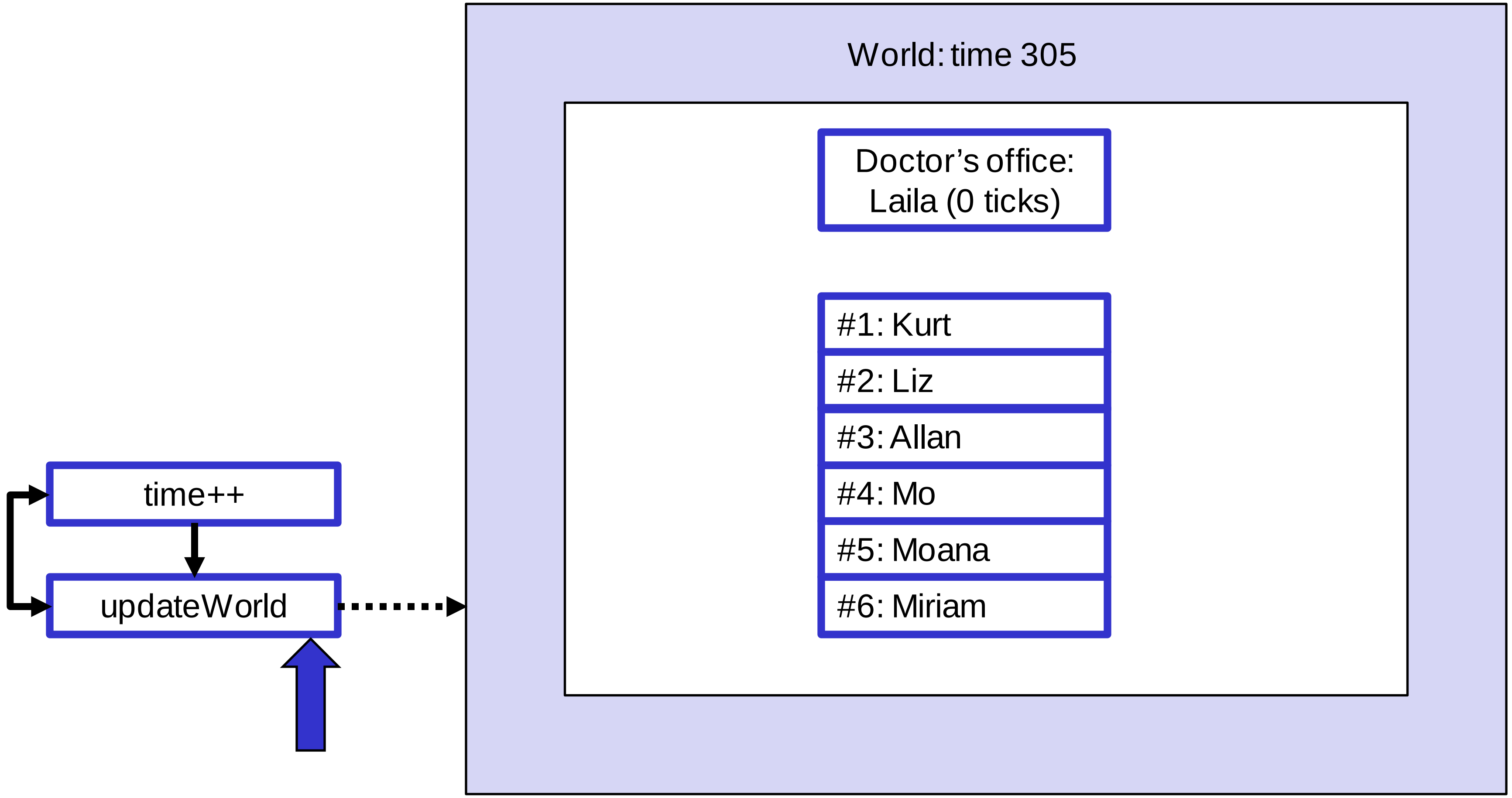
# Simulation of Queues

- Doctor’s waiting room



# Simulation of Queues

- Doctor’s waiting room



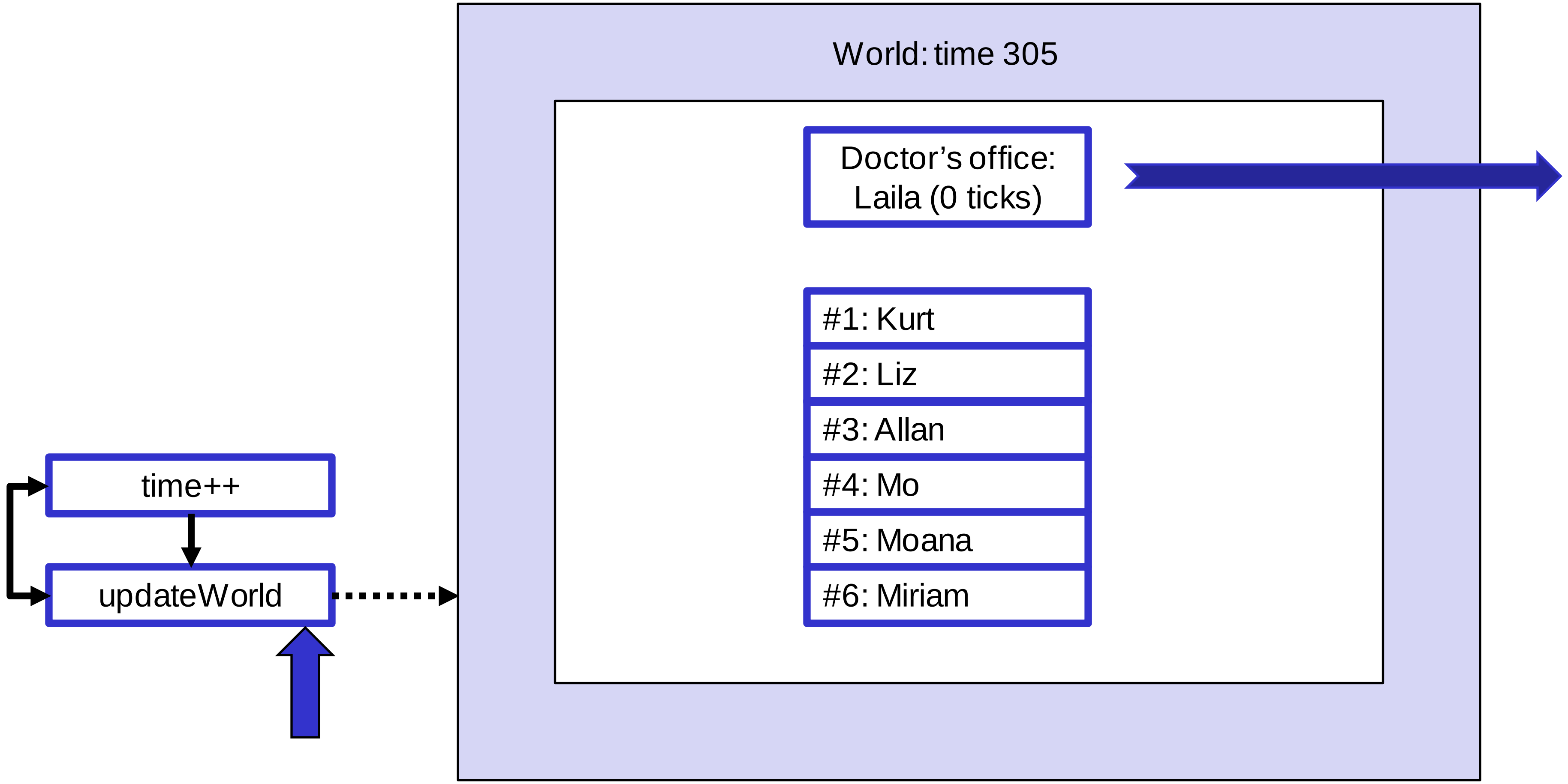
World: time 305

Doctor’s office:
Laila (0 ticks)

#1: Kurt
#2: Liz
#3: Allan
#4: Mo
#5: Moana
#6: Miriam

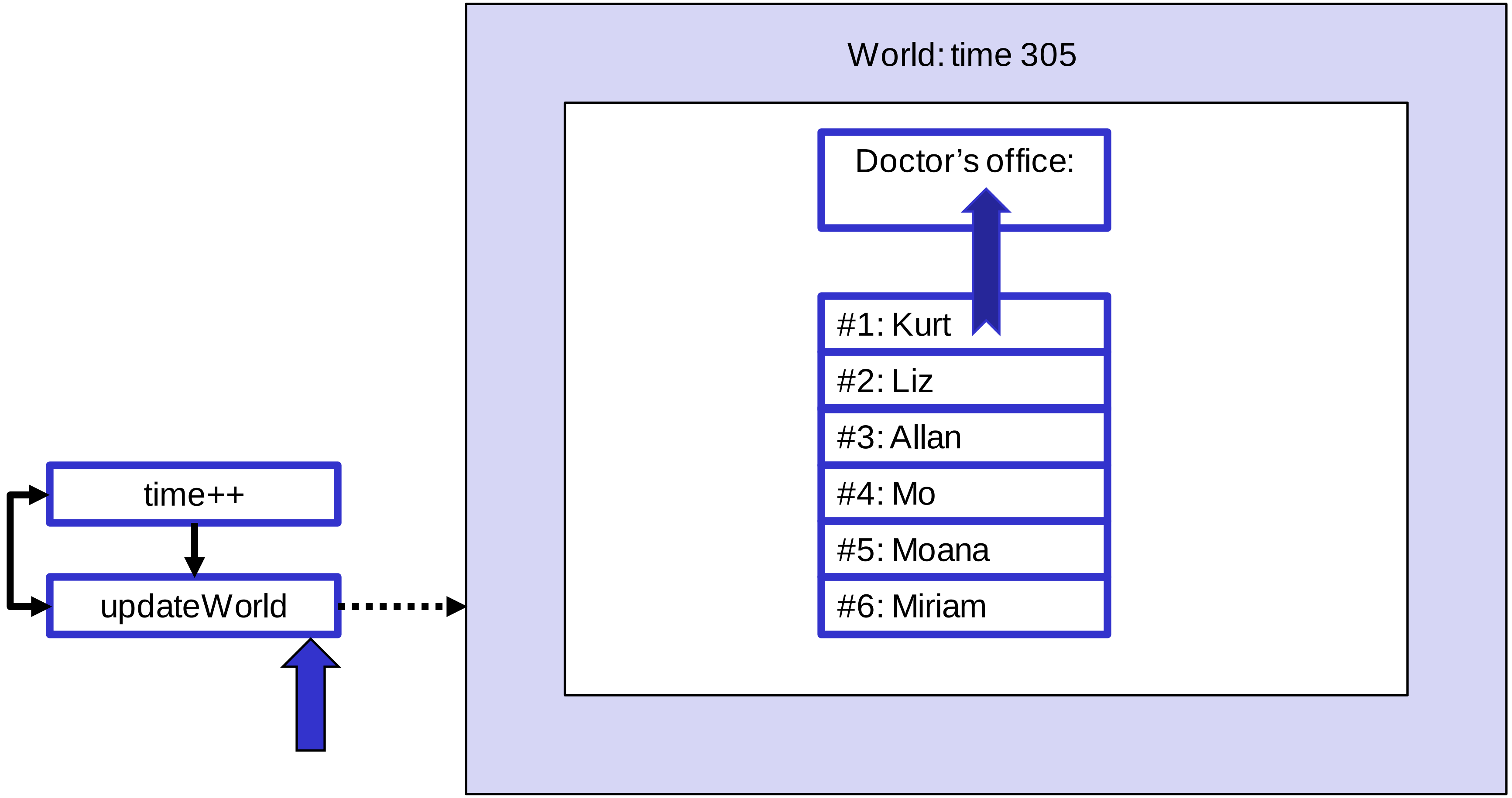
# Simulation of Queues

- Doctor’s waiting room



# Simulation of Queues

- Doctor’s waiting room



time++

updateWorld

World: time 305

Doctor’s office:

#1: Kurt

#2: Liz

#3: Allan

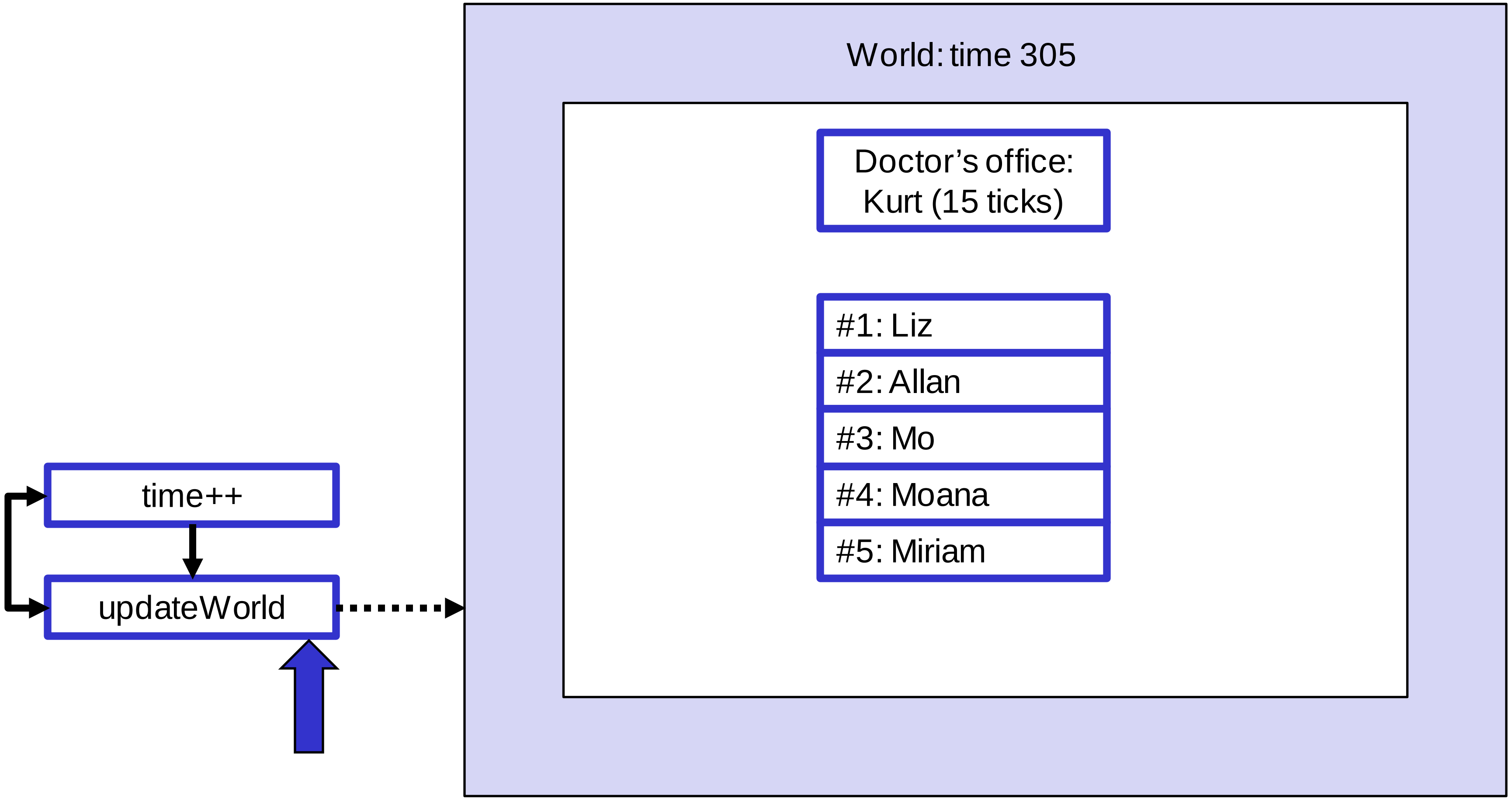
#4: Mo

#5: Moana

#6: Miriam

# Simulation of Queues

- Doctor’s waiting room



time++

updateWorld

World: time 305

Doctor’s office:
Kurt (15 ticks)

#1: Liz
#2: Allan
#3: Mo
#4: Moana
#5: Miriam

# Simulation

- Event-based simulation
  - Keep a (priority) queue of all the events that are going to happen
  - Each iteration of the main loop
    - takes the first event off the queue,
    - updates all entities affected by the event,
    - adds new events to the queue for each future consequence/effect of this event.
  - More efficient if most entities don’t change most of the time
    but conceptually more complicated