
COMP 322: Fundamentals of Parallel Programming

Lecture 23: Actors and Places, Linearizability of Concurrent Objects

Vivek Sarkar

Department of Computer Science, Rice University
vsarkar@rice.edu

<https://wiki.rice.edu/confluence/display/PARPROG/COMP322>



Acknowledgments for Today's Lecture

- Maurice Herlihy and Nir Shavit. The art of multiprocessor programming. Morgan Kaufmann, 2008.
 - Optional text for COMP 322
 - Slides and code examples extracted from <http://www.elsevierdirect.com/companion.jsp?ISBN=9780123705914>



Outline

- Actors and Places
- Linearizability of Concurrent Objects



Places in HJ (Recap)

`import hj.lang.place;`

`here` = place at which current task is executing

`place.MAX_PLACES` = total number of places (runtime constant)

Specified by value of `p` in runtime option, `-places p:w`

`place.factory.place(i)` = place corresponding to index `i`

`<place-expr>.toString()` returns a string of the form "place(id=0)"

`<place-expr>.id` returns the id of the place as an int

`<place-expr>.next()` returns the next place

= `place.factory.place((<place-expr>.id + 1) % place.MAX_PLACES)`

`async at(P) S`

- Creates new task to execute statement `S` at place `P`
- `async S` is equivalent to `async at(here) S`
- Main program task starts at `place.factory.place(0)`



Actors in HJ (Recap)

- Create your custom class which extends `hj.lang.Actor<Object>` ,and implement the void `process()` method

```
import hj.lang.Actor;
class MyActor extends Actor<Object> {
    protected void process(Object message) {
        System.out.println("Processing " + message);
    }
}
```

- Instantiate and start your actor

```
Actor<Object> anActor = new MyActor();
anActor.start(); //Start actor at same place as parent task
```

- Send messages to the actor

```
anActor.send(aMessage); //aMessage can be any object in general
```

- Call `exit()` to terminate an actor

```
protected void process(Object message) {
    if (message.someCondition()) exit();
}
```

- Actor execution implemented as async tasks in HJ



Adding support for places in HJ actors

- Basic approach: include an **optional place parameter** in the `start()` method

```
Actor<Object> anActor = new MyActor();
```

```
anActor.start(p);    // Start actor at place p
```

- Example:

```
SievePlaceActor nextActor = new SievePlaceActor(...);
```

```
// Start actor at next place, relative to current place  
nextActor.start(here.next());
```



Outline

- Actors and Places
- Linearizability of Concurrent Objects



Concurrent Objects

- A concurrent object is an object that can correctly handle methods invoked in parallel by different tasks or threads
 - Originated as monitors
 - Also referred to as “thread-safe objects”
- For simplicity, it is usually assumed that the body of each method in a concurrent object is itself sequential
 - Assume that method does not create child async tasks
- Implementations of methods can be serial as in monitors (e.g., enclose each method in an object-based isolated statement) or concurrent (e.g., `ConcurrentHashMap`, `ConcurrentLinkedQueue` and `CopyOnWriteArraySet`)
- A desirable goal is to develop implementations that are concurrent while being as close to the semantics of the serial version as possible

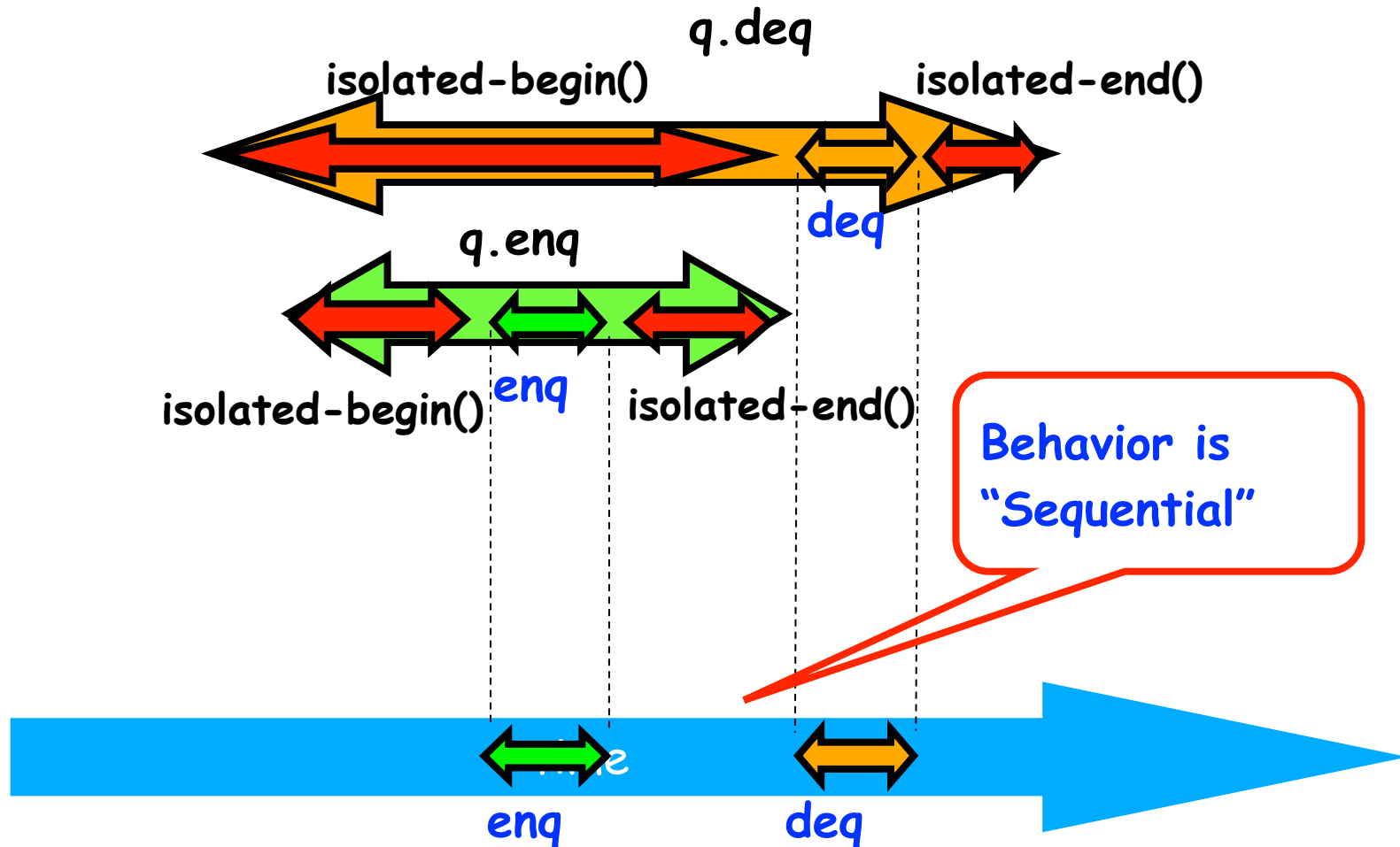


The Big Question!

- Consider a simple FIFO (First In, First Out) queue as a canonical example of a concurrent object
 - Method `q.enq(o)` inserts object `o` at the tail of the queue
 - Assume that there is unbounded space available for all `enq()` operations to succeed
 - Method `q.deq()` removes and returns the item at the head of the queue.
 - Throws `EmptyException` if the queue is empty.
- What does it **mean** for a concurrent object like a FIFO queue to be correct?
 - What is a concurrent FIFO queue?
 - FIFO means strict temporal order
 - Concurrent means ambiguous temporal order



Describing the concurrent via the sequential



Source: http://www.elsevierdirect.com/companions/9780123705914/Lecture%20Slides/03~Chapter_03.ppt



Informal definition of Linearizability

- A linearizable execution is one in which the semantics of a set of method calls performed in parallel on a concurrent object is equivalent to that of some legal linear sequence of those method calls.
- A linearizable concurrent object is one for which all possible executions are linearizable.



Table 1: Example execution of a monitor-based implementation of FIFO queue q

Is this a linearizable execution?

Time	Task A	Task B
0	Invoke $q.enq(x)$	
1	Work on $q.enq(x)$	
2	Work on $q.enq(x)$	
3	Return from $q.enq(x)$	
4		Invoke $q.enq(y)$
5		Work on $q.enq(y)$
6		Work on $q.enq(y)$
7		Return from $q.enq(y)$
8		Invoke $q.deq()$
9		Return x from $q.deq()$

Yes! Equivalent to " $q.enq(x) ; q.enq(y) ; q.deq():x$ "



Table 2: Example execution of method calls on a concurrent FIFO queue q

Is this a linearizable execution?

Time	Task A	Task B
0	Invoke $q.enq(x)$	
1	Work on $q.enq(x)$	Invoke $q.enq(y)$
2	Work on $q.enq(x)$	Return from $q.enq(y)$
3	Return from $q.enq(x)$	
4		Invoke $q.deq()$
5		Return x from $q.deq()$

Yes! Equivalent to " $q.enq(x) ; q.enq(y) ; q.deq():x$ "



Table 3: Example of a non-linearizable execution on a concurrent FIFO queue q

Is this a linearizable execution?

Time	Task A	Task B
0	Invoke $q.enq(x)$	
1	Return from $q.enq(x)$	
2		Invoke $q.enq(y)$
3	Invoke $q.deq()$	Work on $q.enq(y)$
4	Work on $q.deq()$	Return from $q.enq(y)$
5	Return y from $q.deq()$	

- No! $q.enq(x)$ must precede $q.enq(y)$ in all linear sequences of method calls invoked on q . It is illegal for the $q.deq()$ operation to return y .



Alternate definition of Linearizability

- Assume that each method call takes effect “instantaneously” at some distinct point in time between its invocation and return.
- Execution is linearizable if we can choose instantaneous points that are consistent with a sequential execution in which methods are executed at those points



Table 2: Example execution of method calls on a concurrent FIFO queue q

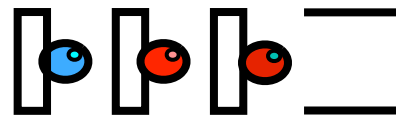
Is this a linearizable execution?

Time	Task A	Task B
0	Invoke $q.enq(x)$	
1	Work on $q.enq(x)$	Invoke $q.enq(y)$
2	Work on $q.enq(x)$	Return from $q.enq(y)$
3	Return from $q.enq(x)$	
4		Invoke $q.deq()$
5		Return x from $q.deq()$

Yes! Equivalent to " $q.enq(x) ; q.enq(y) ; q.deq():x$ "



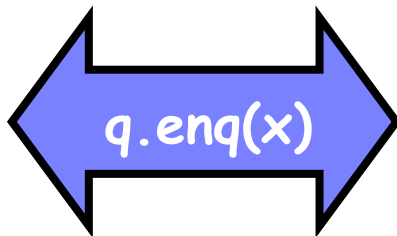
An Example



Source: http://www.elsevierdirect.com/companions/9780123705914/Lecture%20Slides/03~Chapter_03.ppt



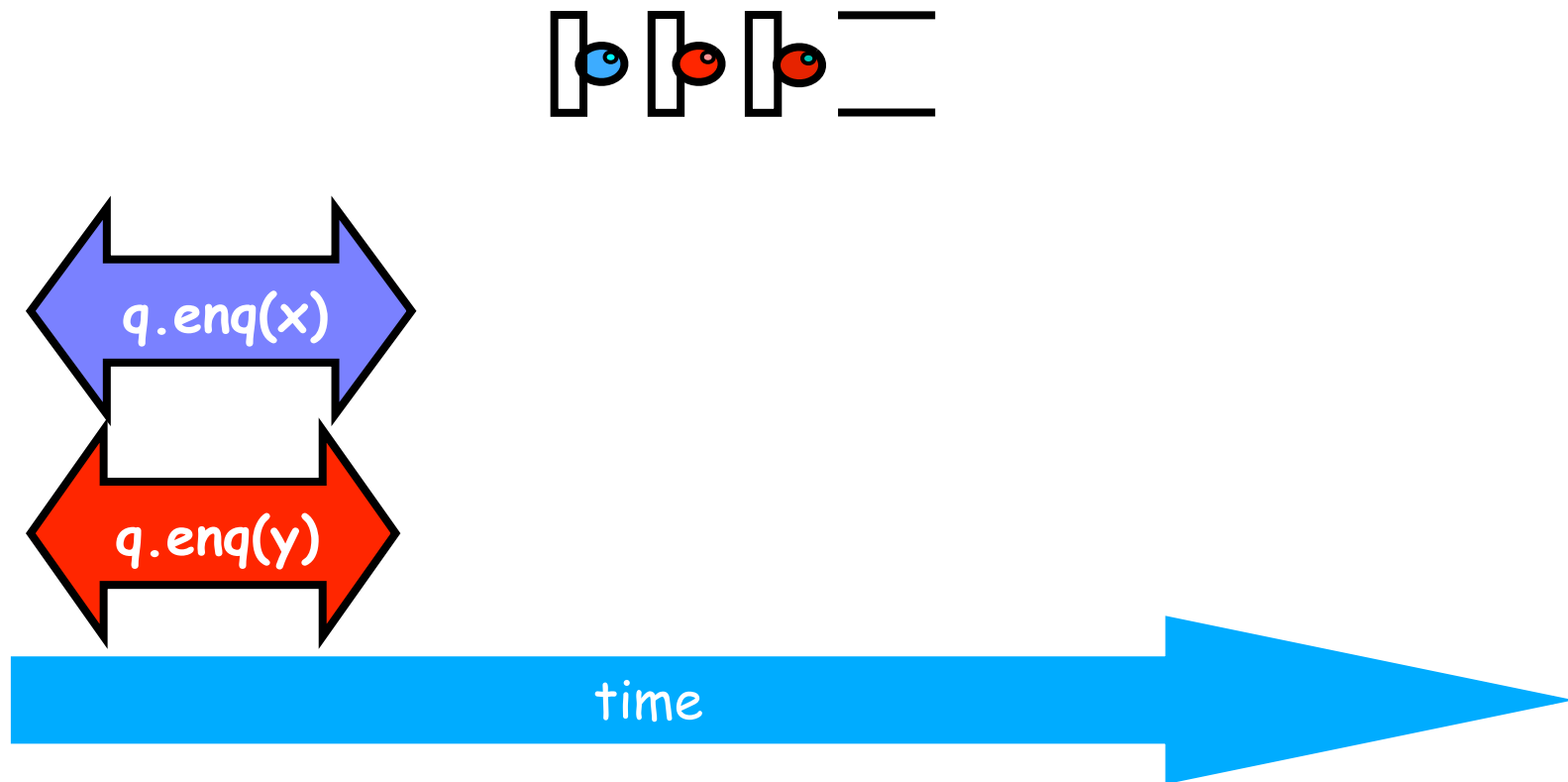
Example



Source: http://www.elsevierdirect.com/companions/9780123705914/Lecture%20Slides/03~Chapter_03.ppt



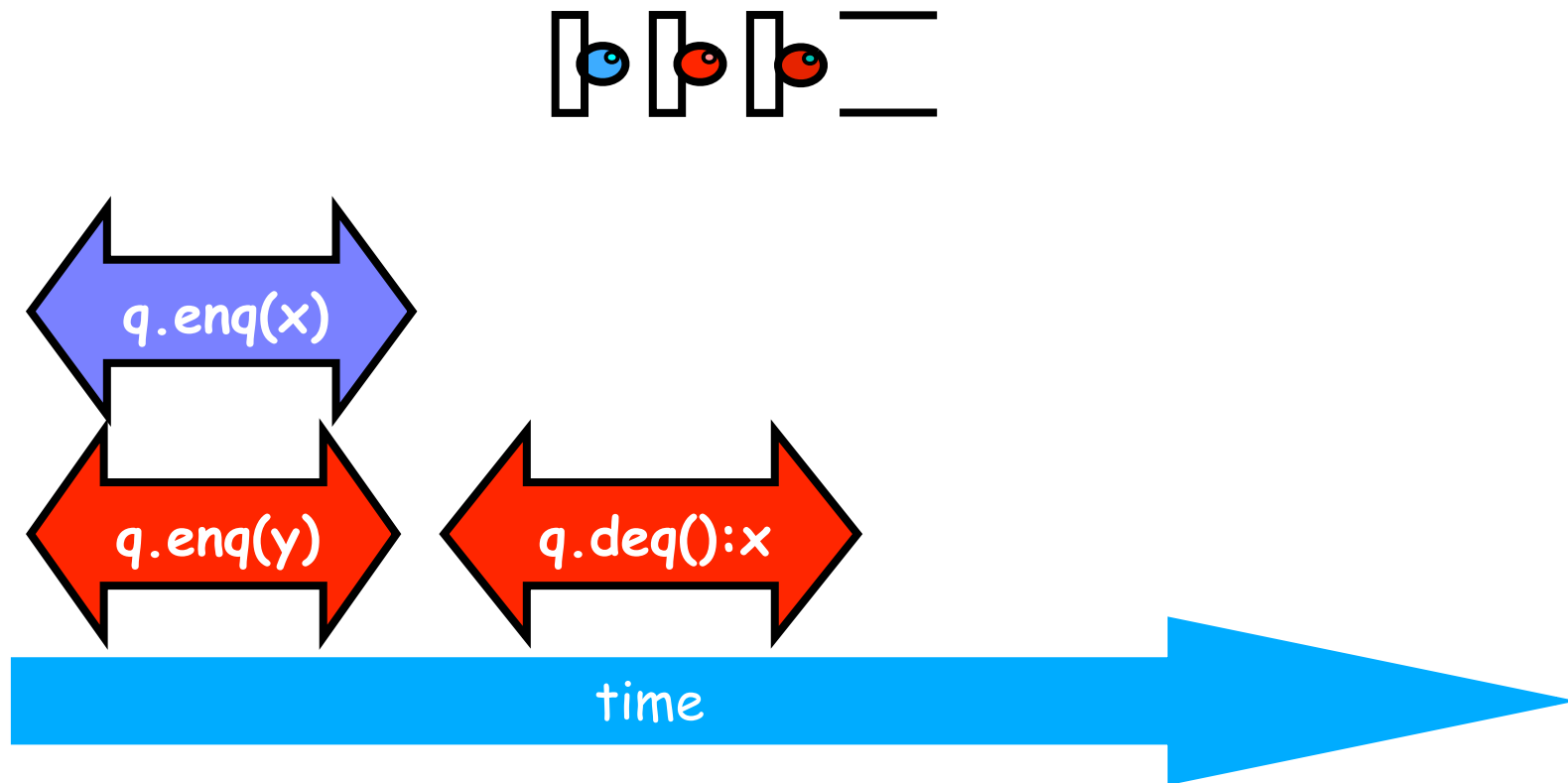
Example



Source: http://www.elsevierdirect.com/companions/9780123705914/Lecture%20Slides/03~Chapter_03.ppt



Example

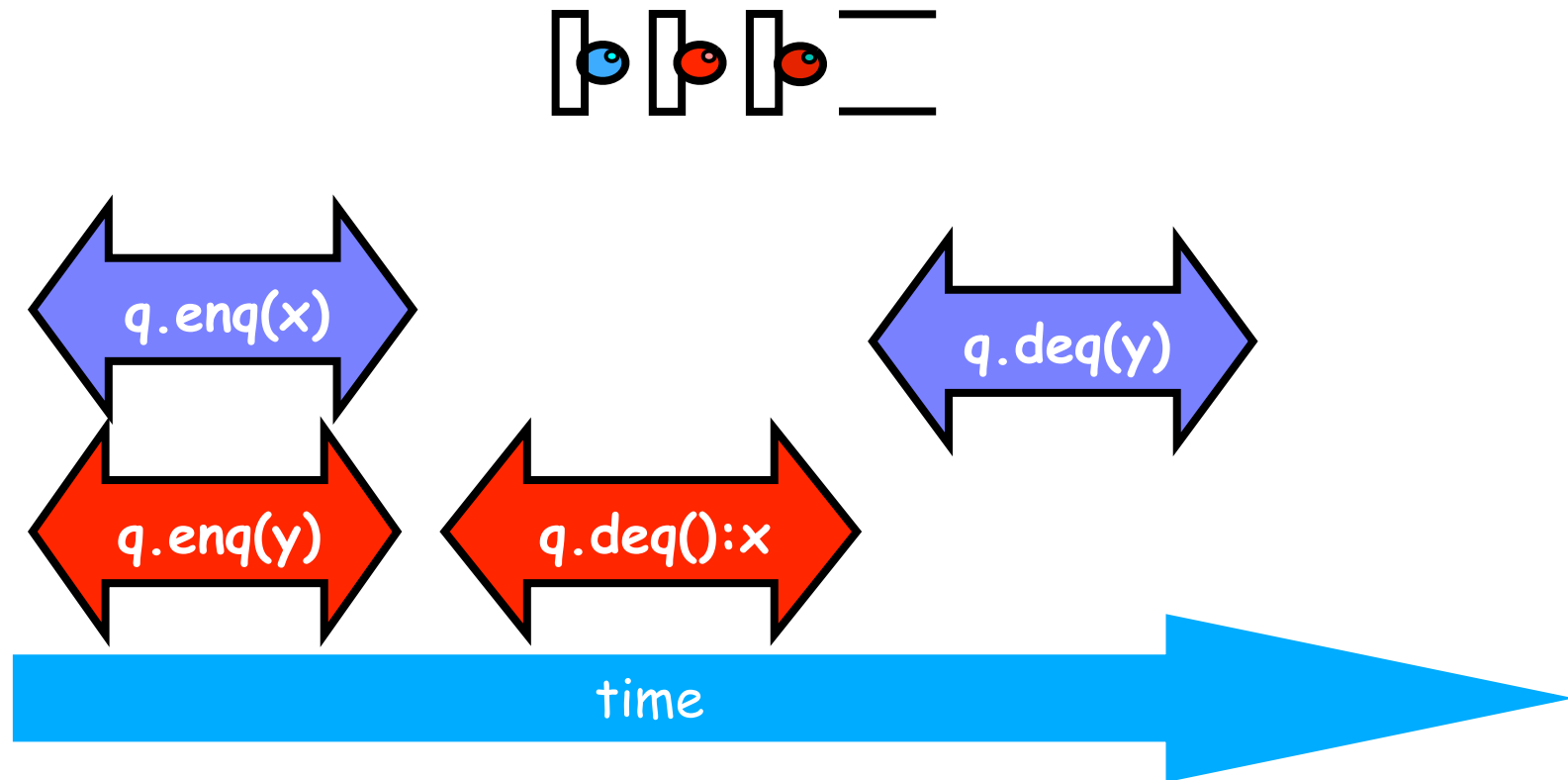


Source: http://www.elsevierdirect.com/companions/9780123705914/Lecture%20Slides/03~Chapter_03.ppt





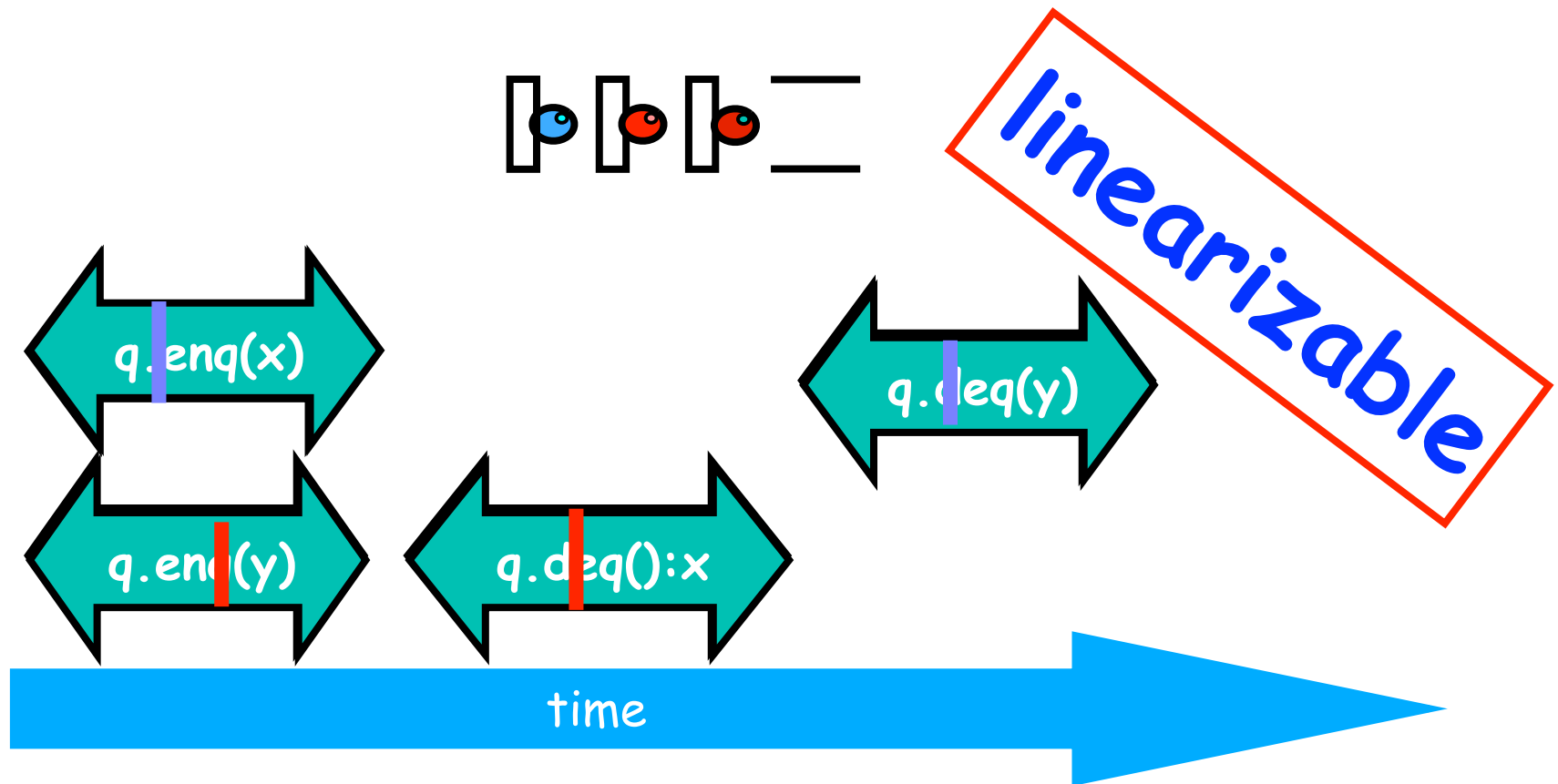
Example



Source: http://www.elsevierdirect.com/companions/9780123705914/Lecture%20Slides/03~Chapter_03.ppt



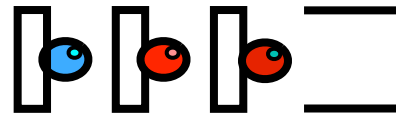
Example



Source: http://www.elsevierdirect.com/companions/9780123705914/Lecture%20Slides/03~Chapter_03.ppt



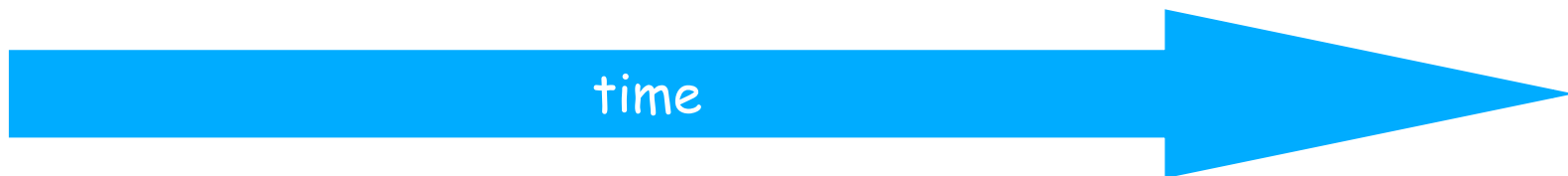
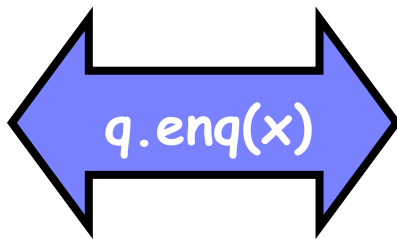
Another Example (like Table 3)



Source: http://www.elsevierdirect.com/companions/9780123705914/Lecture%20Slides/03~Chapter_03.ppt



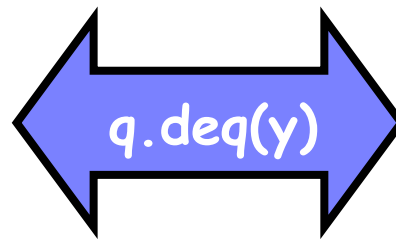
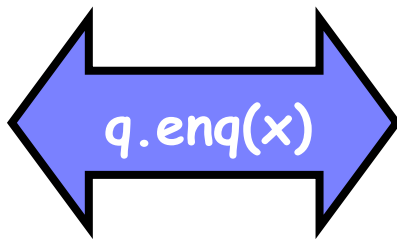
Another Example



Source: http://www.elsevierdirect.com/companions/9780123705914/Lecture%20Slides/03~Chapter_03.ppt



Another Example

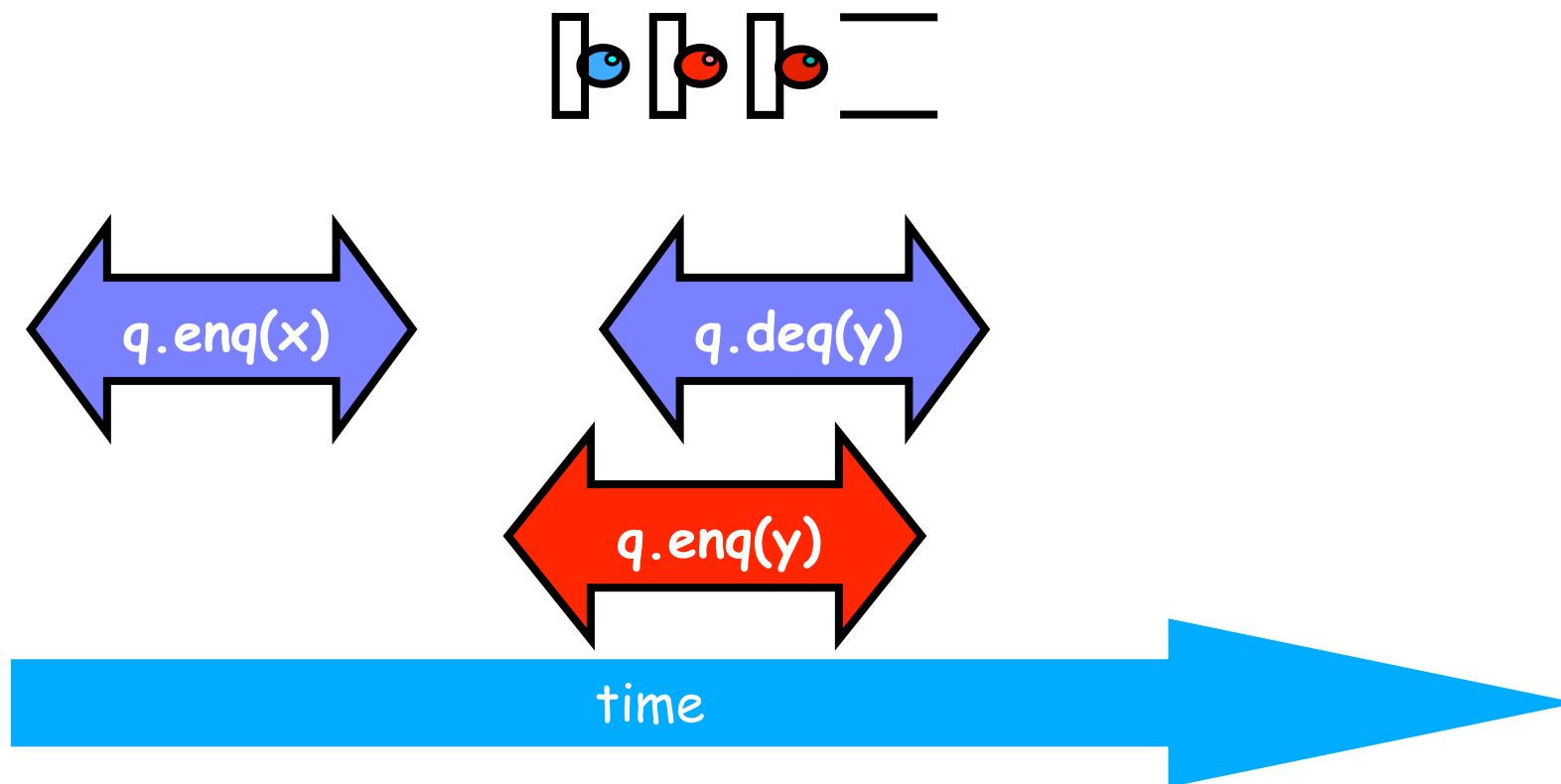


Source: http://www.elsevierdirect.com/companions/9780123705914/Lecture%20Slides/03~Chapter_03.ppt





Another Example

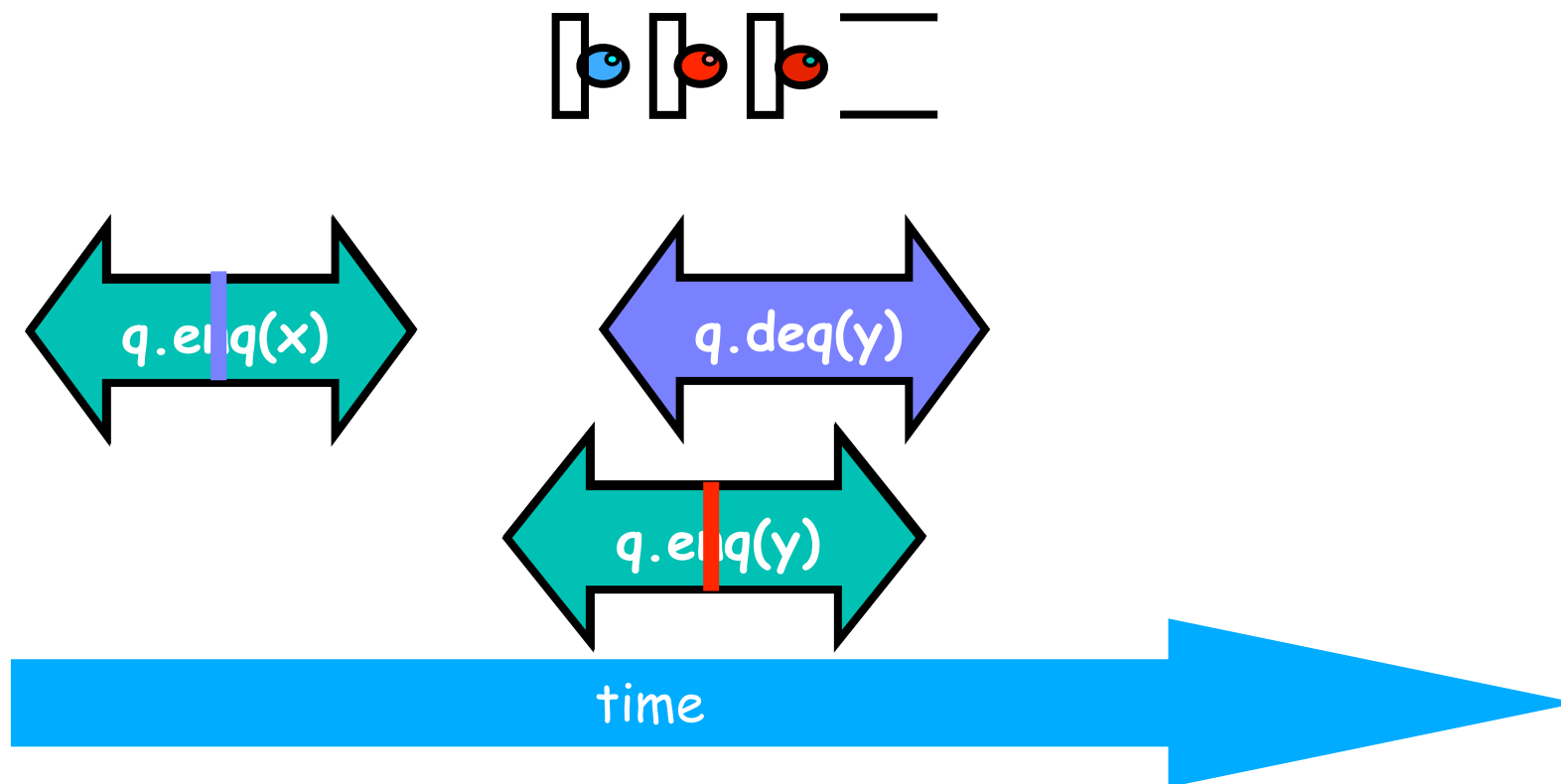


Source: http://www.elsevierdirect.com/companions/9780123705914/Lecture%20Slides/03~Chapter_03.ppt





Another Example

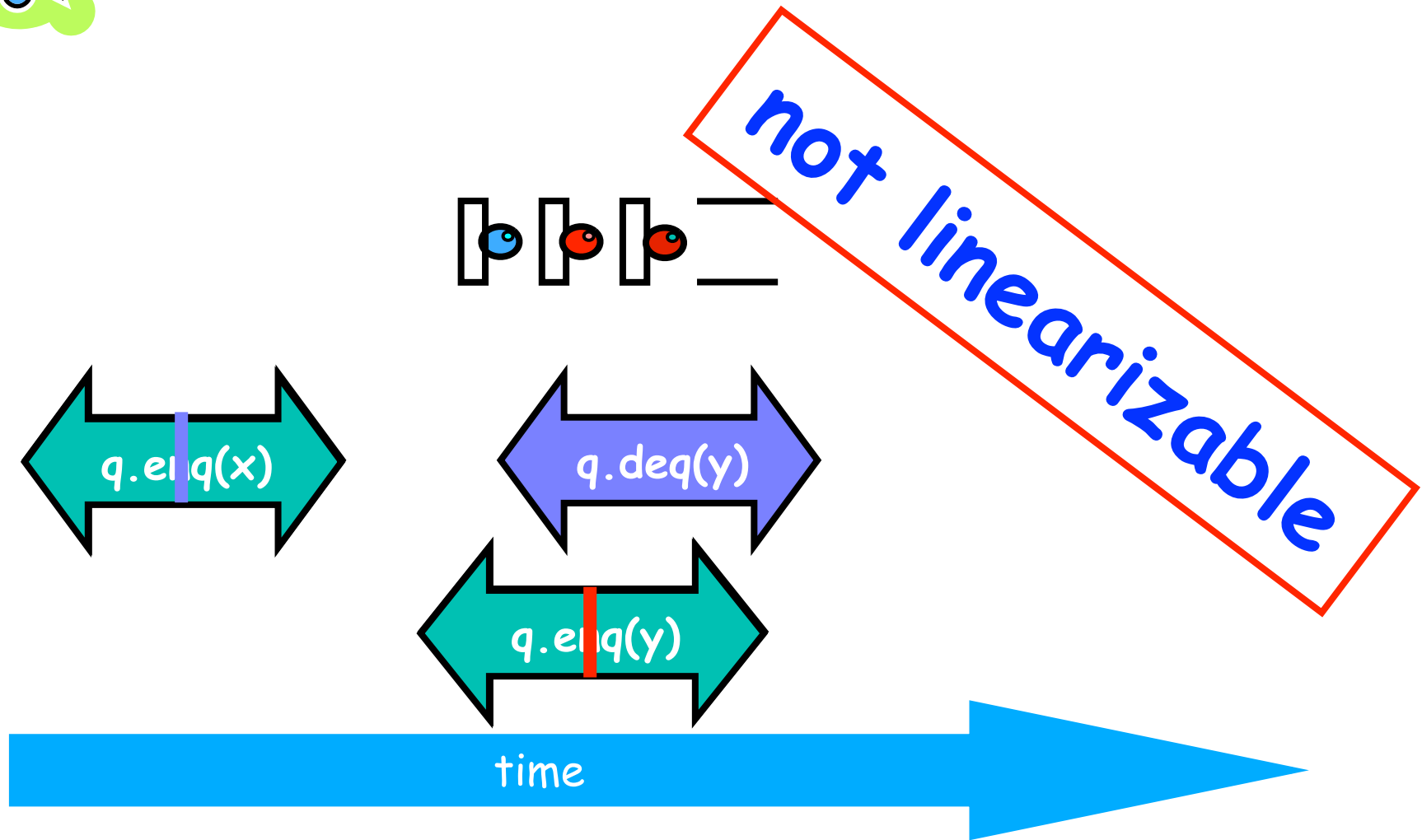


Source: http://www.elsevierdirect.com/companions/9780123705914/Lecture%20Slides/03~Chapter_03.ppt





Another Example



Source: http://www.elsevierdirect.com/companions/9780123705914/Lecture%20Slides/03~Chapter_03.ppt



Figure 1: Computation Graph for monitor-based implementation of FIFO queue (Table 1)

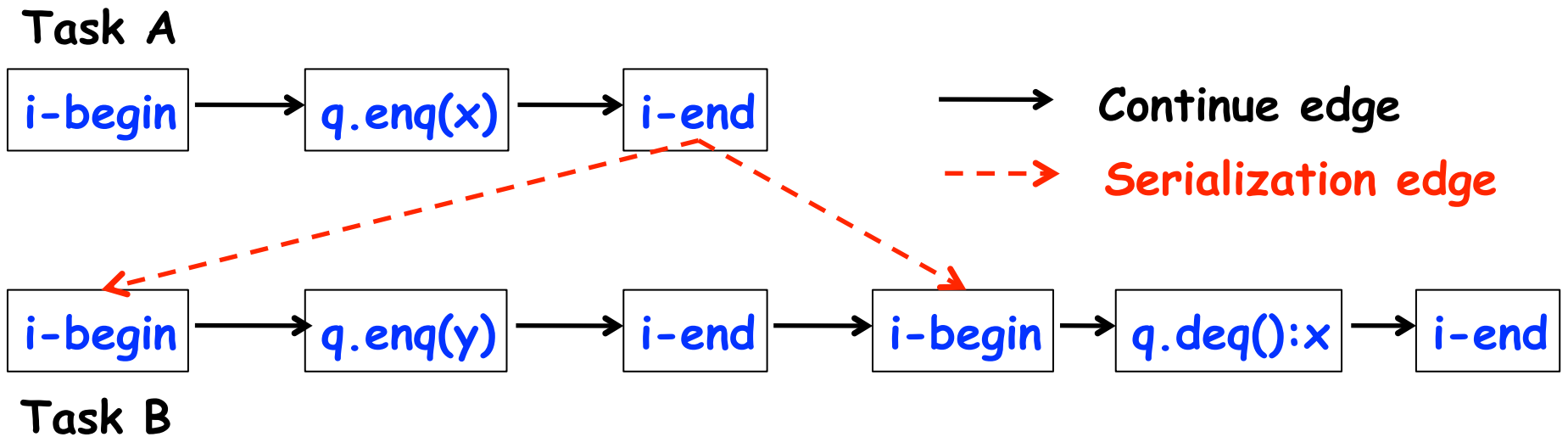
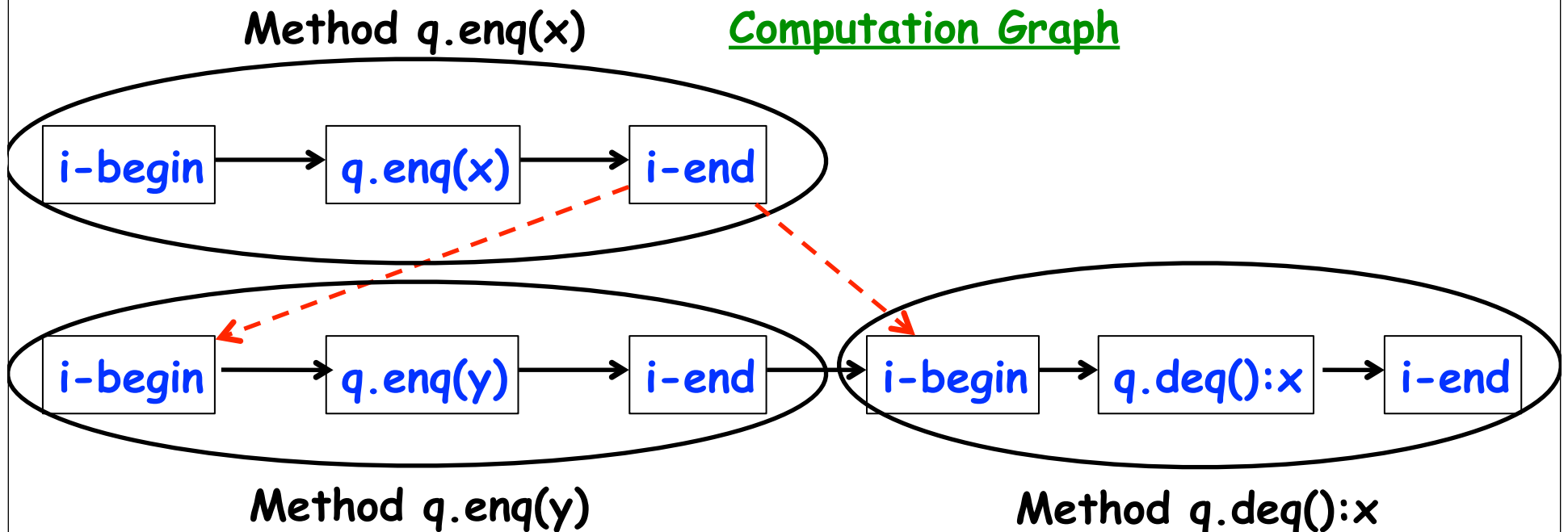
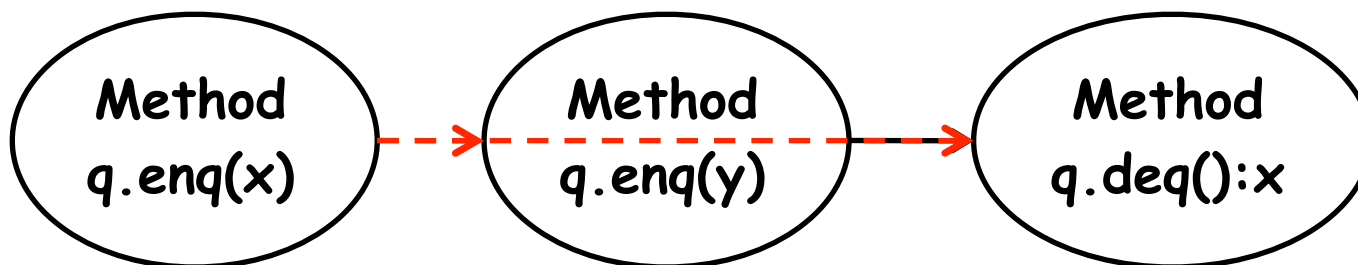


Figure 2: Creating a Reduced Graph to model Instantaneous Execution of Methods



Method-level Reduced Graph



Relating Linearizability to the Computation Graph model

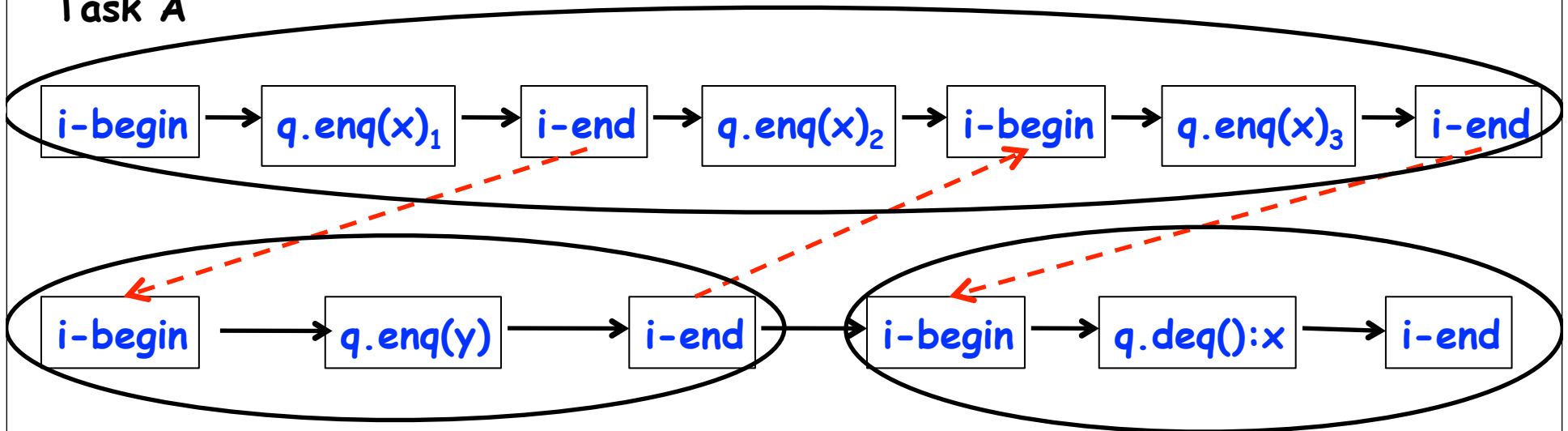
- Given a reduced CG , a sufficient condition for linearizability is that the reduced CG is acyclic as in Figure 2.
- This means that if the reduced CG is acyclic, then the underlying execution must be linearizable.
- However, the converse is not necessarily true, as we will see later.



Figure 3: Example Computation Graph for concurrent implementation of FIFO queue (Table 2)

Computation Graph

Task A



Task B

→ Continue edge - - - -> Serialization edge

Method-level Reduced Graph

