COMP 322: Fundamentals of Parallel Programming

Lecture 24: Linearizability of Concurrent Objects (contd)

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
  - Chapter 3 slides extracted from http://www.elsevierdirect.com/companion.jsp?ISBN=9780123705914
- Lecture on "Linearizability" by Mila Oren
  - http://www.cs.tau.ac.il/~afek/Mila.Linearizability.ppt

Linearizability of Concurrent Objects (Recap)

<u>Concurrent object</u>

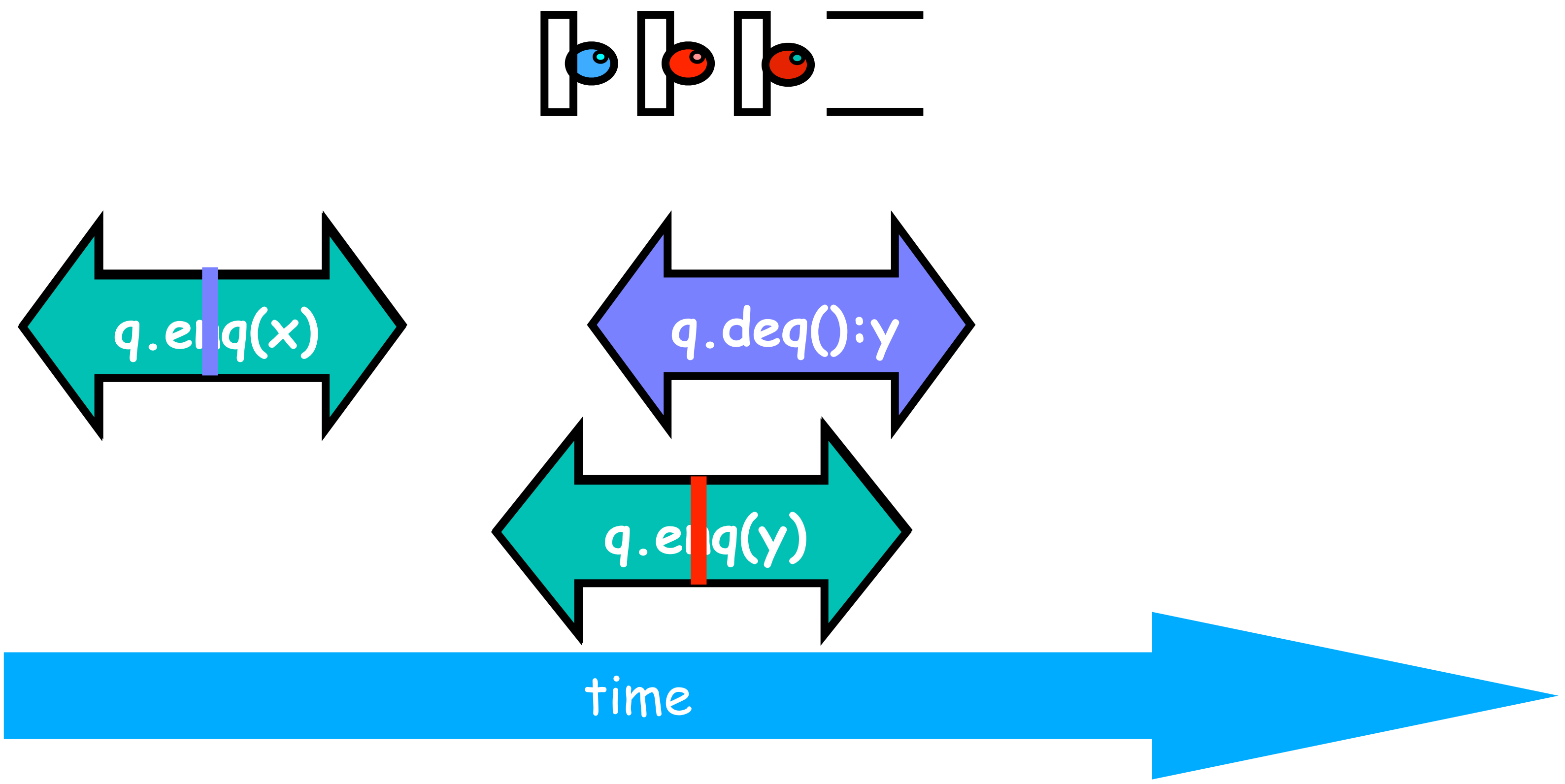
- A concurrent object is an object that can correctly handle methods invoked in parallel by/in different tasks or threads
  - Examples: concurrent queue, AtomicInteger

<u>Linearizability</u>

- Assume that each method call takes effect "instantaneously" at some distinct point in time between its invocation and return.
- An <u>execution</u> is linearizable if we can choose instantaneous points that are consistent with a sequential execution in which methods are executed at those points
- An <u>object</u> is linearizable if all its possible executions are linearizable

Example 1

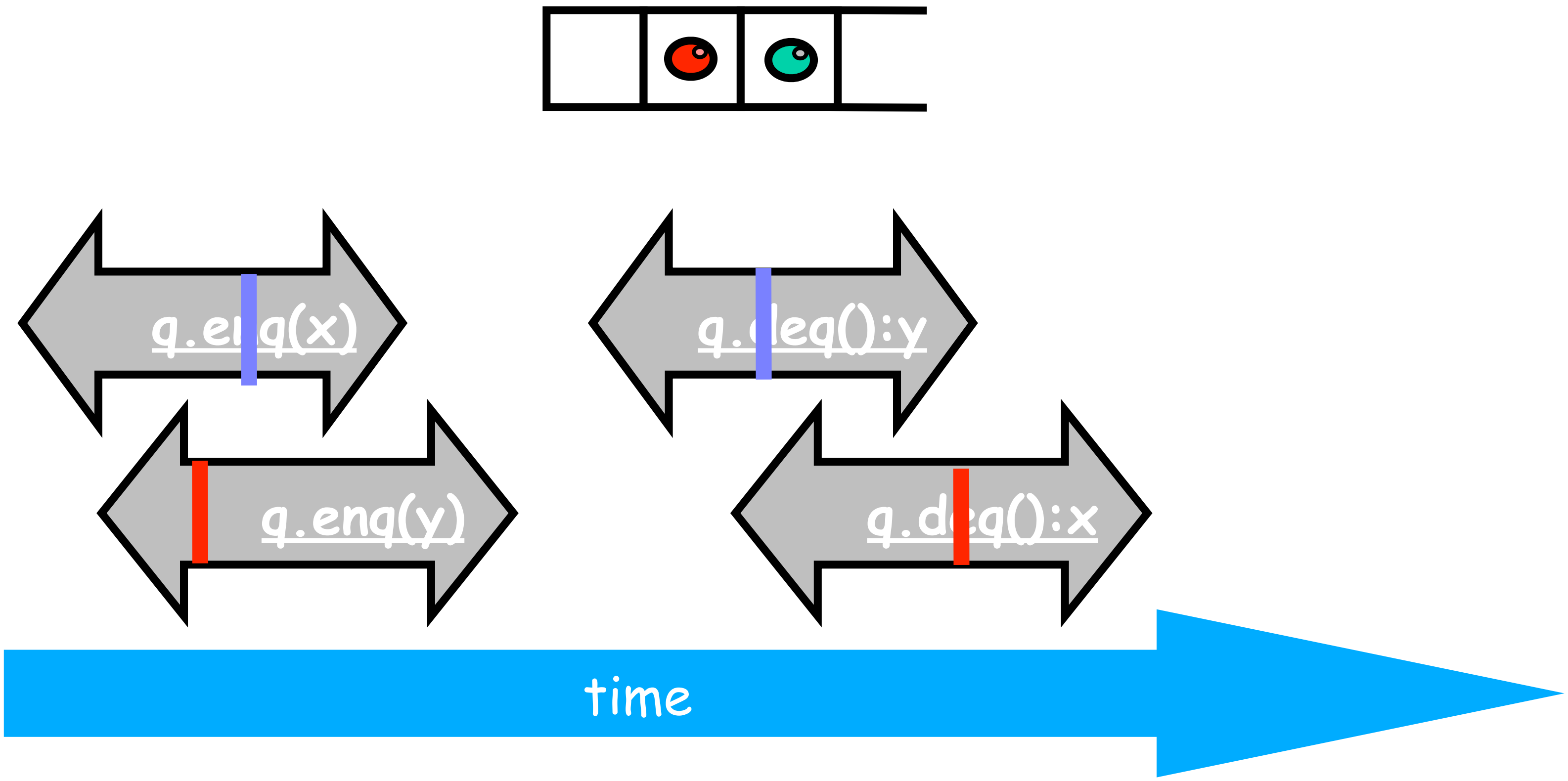
Is this execution linearizable?



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

Example 2

Is this execution linearizable?



One Possible Attempt to Implement a Concurrent Queue

```
// Assume that no. of enq() operations is < Integer.MAX_VALUE
class Queue1 {
  AtomicInteger head = new AtomicInteger(0);
  AtomicInteger tail = new AtomicInteger(0);
  Object[] items = new Object[Integer.MAX_VALUE];
  public void enq(Object x) {
    int slot = tail.getAndIncrement(); // isolated(tail) ...
   items[slot] = x;
 } // enq
  public Object deq() throws EmptyException {
    int slot = head.getAndIncrement(); // isolated(head) ...
    Object value = items[slot];
    if (value == null) throw new EmptyException();
    return value;
  } // deq
} // Queue1

// Client code
finish {
  Queue1 q = new Queue1();
  async q.enq(new Integer(1));
  q.enq(newInteger(2));
  Integer x = (Integer) q.deq();
}
```

Is there a possible execution for which deq() results in an EmptyException?

Formalizing Linearizability

- We split a method call into two events:
  - Invocation: method names + args
    - q.enq(x)
  - Response: result or exception
    - q.enq(x) returns void
    - q.deq() returns x or throws emptyException

Notations for invocations and responses

- Invocation notation: A q.enq(x)
  - A – thread
  - q – object
  - enq – method
  - x – arg
- Response notation: A q: void , A q: empty()
  - A – thread
  - q – object
  - void – result, exception

Execution History

A sequence of invocations and responses. It describes an execution.

H =

A q.enq(3)
A q:void
A q.enq(5)
B p.enq(4)
B p:void
B q.deq()
B q:3

Some Definitions

- Invocation and Response match if:

thread names and object names agree.

A q.enq(3)
A q:void

- A pending invocation is an invocation that has no matching response.

A q.enq(3)
A q:void
A q.enq(5)
B q.deq()
B q:3

- Complete history: history without pending invocations.

Sequential History

- Sequential history: A sequence of matches, can end with pending invocation.

A q.enq(3)
A q:void
B p.enq(4)
B p:void
B q.deq()
B q:3
A q:enq(5)

Object and Thread Projections of Histories

- Object projection:

  H | q=

  A q.enq(3)
  A q:void
  A q.enq(5)
  B q.deq()
  B q:3

- Thread projection:

  H | A =

  A q.enq(3)
  A q:void
  A q.enq(5)

Well-formed and Equivalent Histories

- Well-formed history: for each thread A, H|A is sequential.
- Equivalent histories: H and G are equivalent if for each thread A: H|A = G|A

H =	A q.enq(3) B p.enq(4) B p:void B q.deq() A q:void B q:3	G =	A q.enq(3) A q:void B p.enq(4) B p:void B q.deq() B q:3

Precedes relation on method calls

- Method call m0 precedes method call m1 in history H if m0's response event precedes m1's invocation event in H

A q.enq(3)
B p.enq(4)
B p.void
A q:void
B q.deq()
B q:3

Notation:

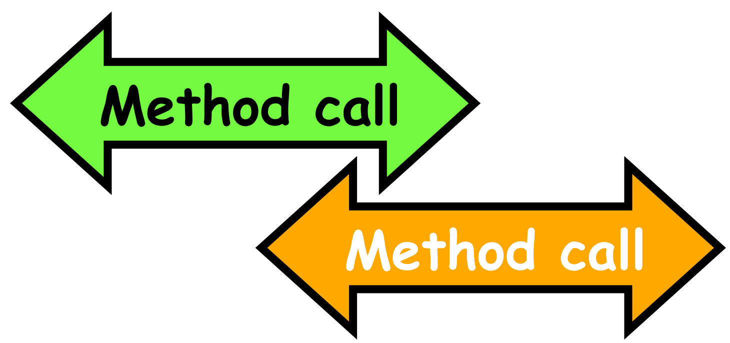
$m_0 \rightarrow_H m_1$

m_0 precedes m_1

Non-Precedence

A q.enq(3)
B p.enq(4)
B p.void
B q.deq()
A q:void
B q:3

Some method calls overlap one another



Legality condition for a sequential history

- A sequential history H is legal if:

  for each object x, H|x is in the sequential specification for x.
- for example: objects like queue, stack

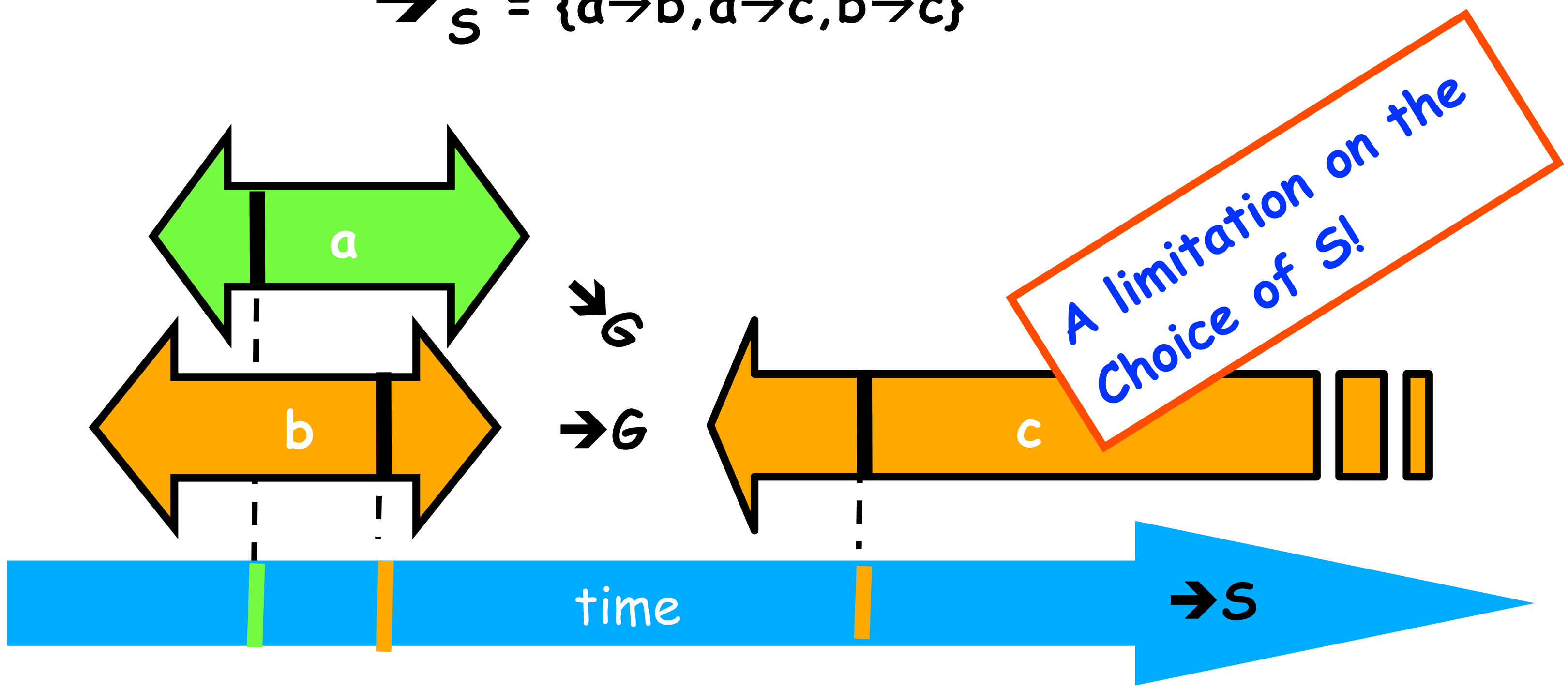
Formal definition of Linearizability

- History H is linearizable if it can be extended to history G so that G is equivalent to legal sequential history S where $\rightarrow_G \subset \rightarrow_S$.
  - if m0 precedes m1 in G, m0 must also precede m1 in S
- G is the same as H but without pending invocations
  - append responses to pending invocations that "took effect"
  - discard pending invocations that "don't matter"

What is meant by $\rightarrow_G \subset \rightarrow_S$

$\rightarrow_G$ = {a→c,b→c}

$\rightarrow_S$ = {a→b,a→c,b→c}

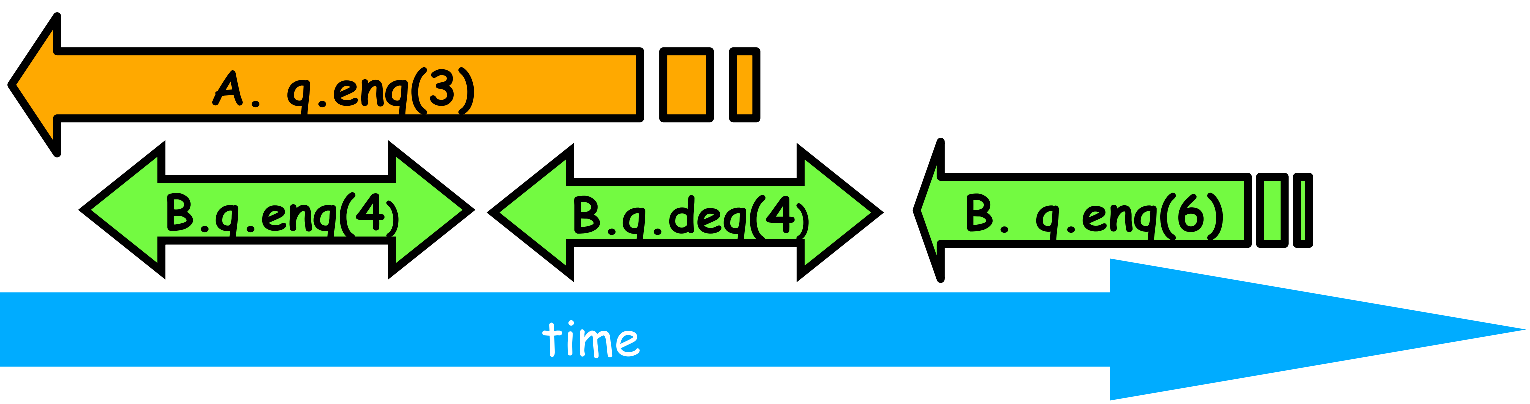


Remarks

- Some pending invocations
  - Took effect, so keep them
  - Discard the rest
- Condition $\rightarrow_G \subset \rightarrow_S$
  - Means that *S* respects "real-time order" of *G*

Example

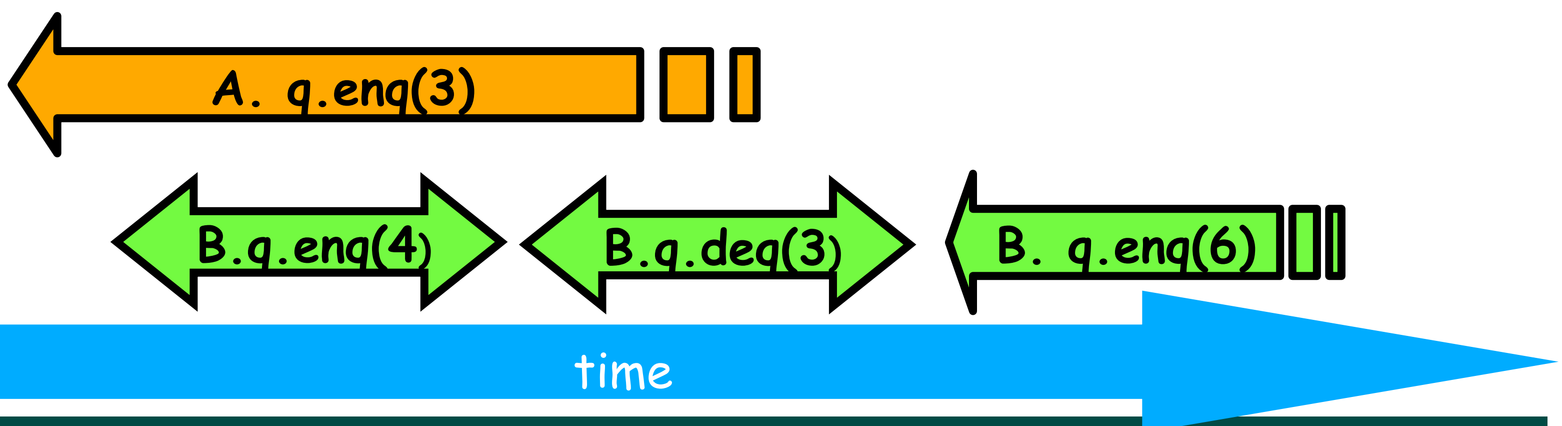
A q.enq(3)
B q.enq(4)
B q:void
B q.deq()
B q:4
B q:enq(6)



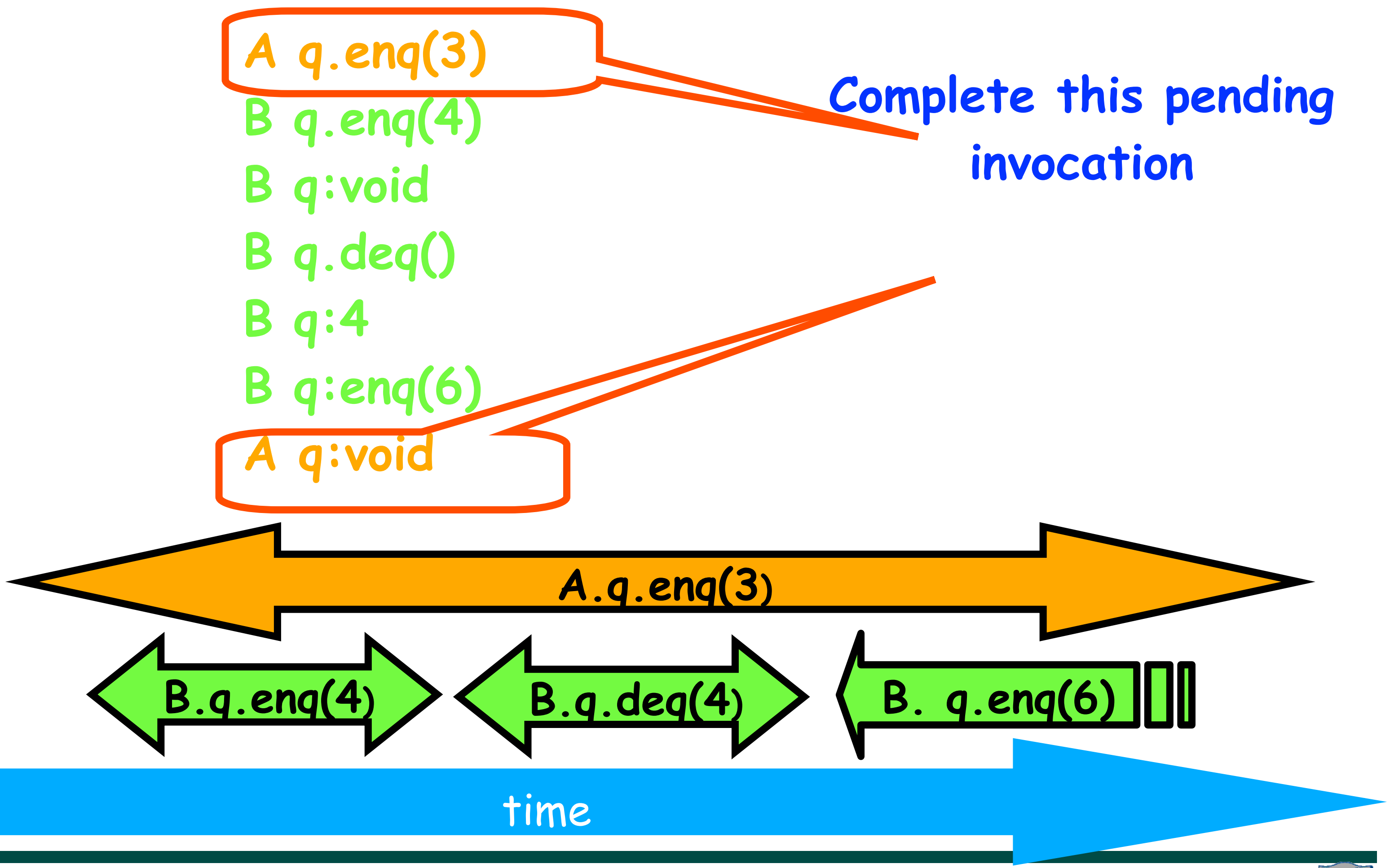
Example

A q.enq(3)
B q.enq(4)
B q:void
B q.deq()
B q:4
B q:enq(6)

Complete this pending invocation



Example

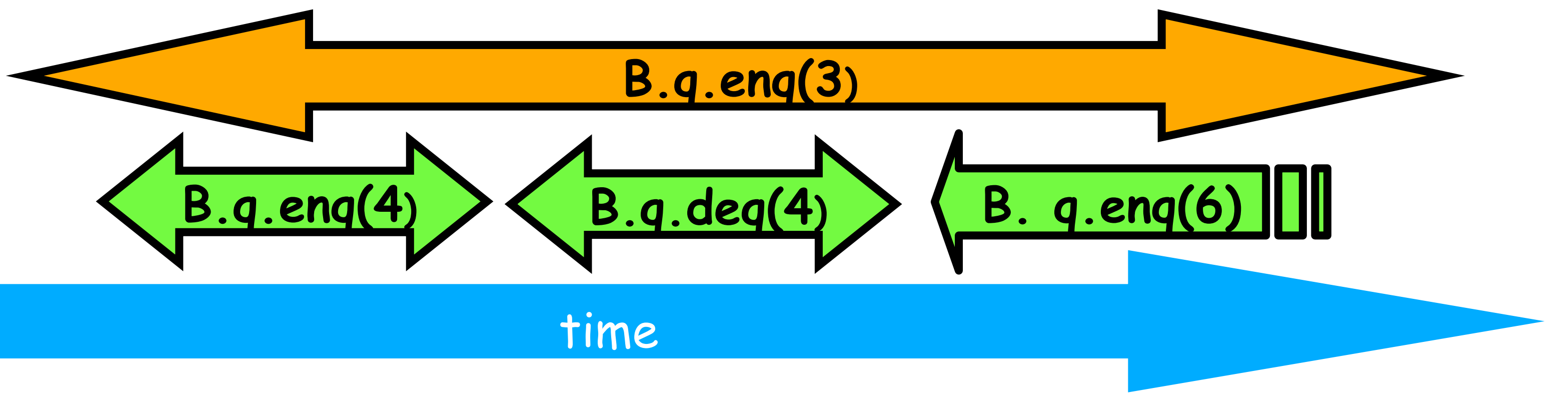


A q.enq(3)
B q.enq(4)
B q:void
B q.deq()
B q:4
B q:enq(6)
A q:void
Complete this pending invocation
A.q.enq(3)
B.q.enq(4)
B.q.deq(4)
B. q.enq(6)
time

Example

A q.enq(3)
B q.enq(4)
B q:void
B q.deq()
B q:4
B q:enq(6)
A q:void

discard this one

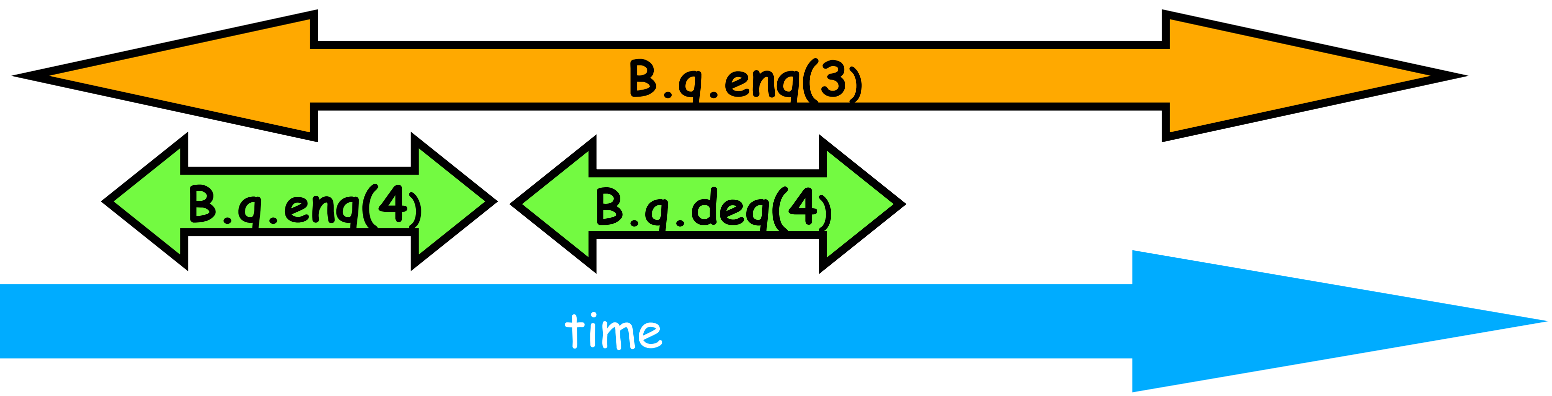


Example

A q.enq(3)
B q.enq(4)
B q:void
B q.deq()
B q:4

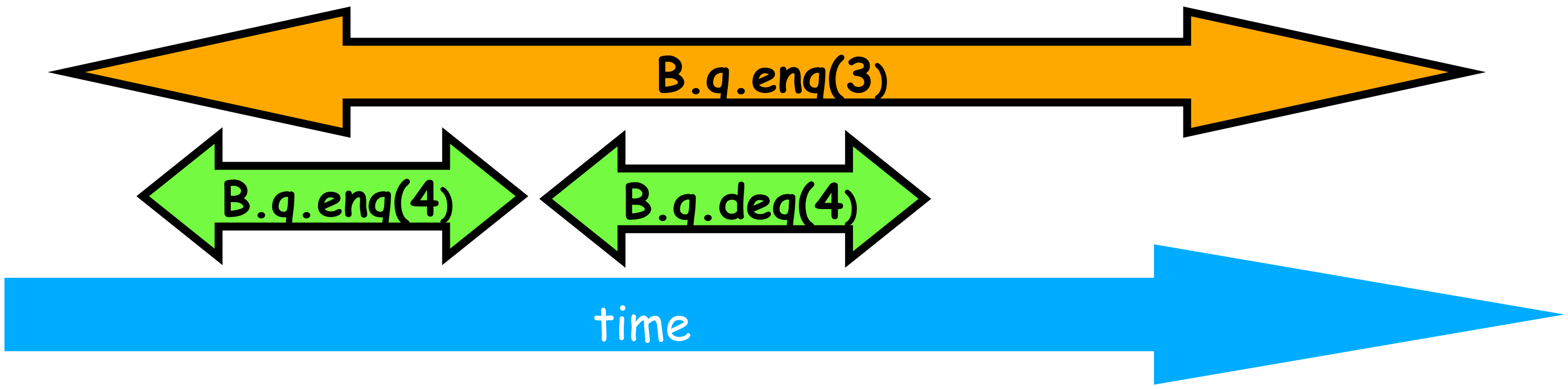
discard this one

A q:void



Example

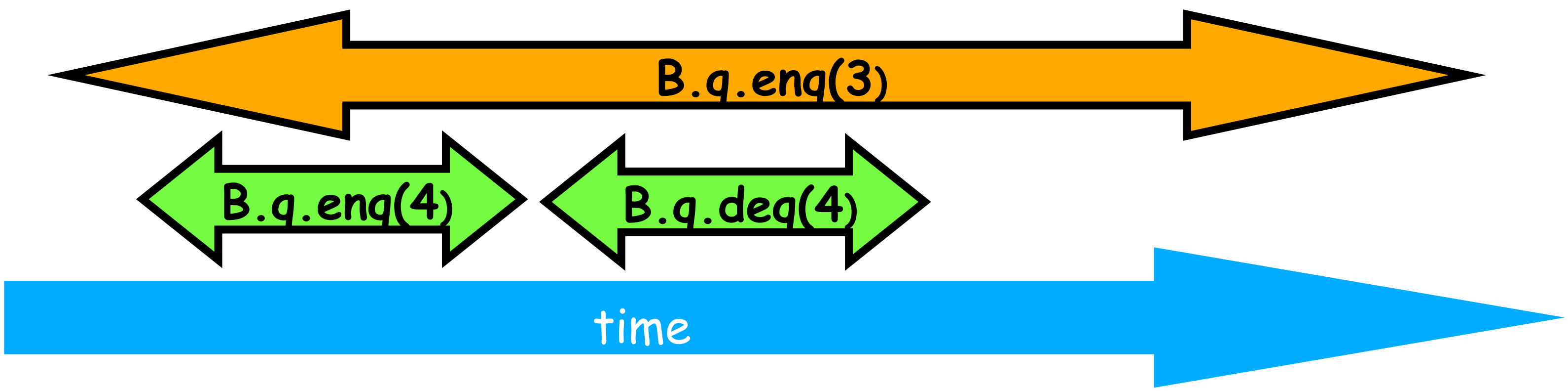
A q.enq(3)
B q.enq(4)
B q:void
B q.deq()
B q:4
A q:void



Example

A q.enq(3)
B q.enq(4)
B q:void
B q.deq()
B q:4
A q:void

B q.enq(4)
B q:void
A q.enq(3)
A q:void
B q.deq()
B q:4

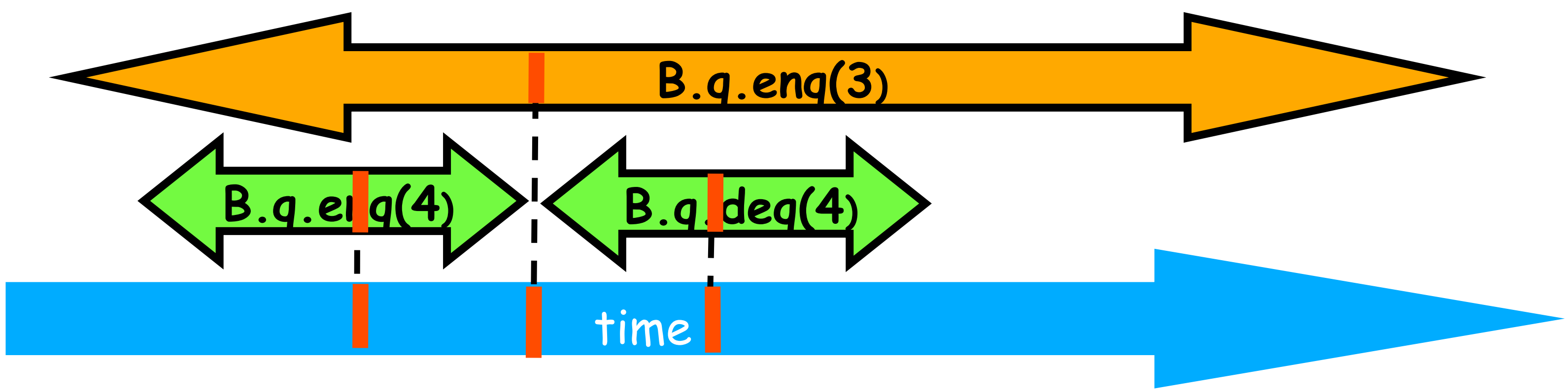


Example

Equivalent sequential history

A q.enq(3)
B q.enq(4)
B q:void
B q.deq()
B q:4
A q:void

B q.enq(4)
B q:void
A q.enq(3)
A q:void
B q.deq()
B q:4



Two Important Properties that follow from Linearizability

1) <u>Composability</u>

- History H is linearizable if and only if
  - For every object x
  - H|x is linearizable
- Why is composability important?
  - Modularity
  - Can prove linearizability of objects in isolation
  - Can compose independently-implemented objects

2) <u>Non-blocking</u>

- one method call is never forced to wait on another
- If method invocation "A q.inv(…)" is pending in history H, then there exists a response "A q:res(…)" such that "H + A q:res(…)" is linearizable