COMP 322: Fundamentals of Parallel Programming

Lecture 23: 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

Actor Life Cycle (Recap)

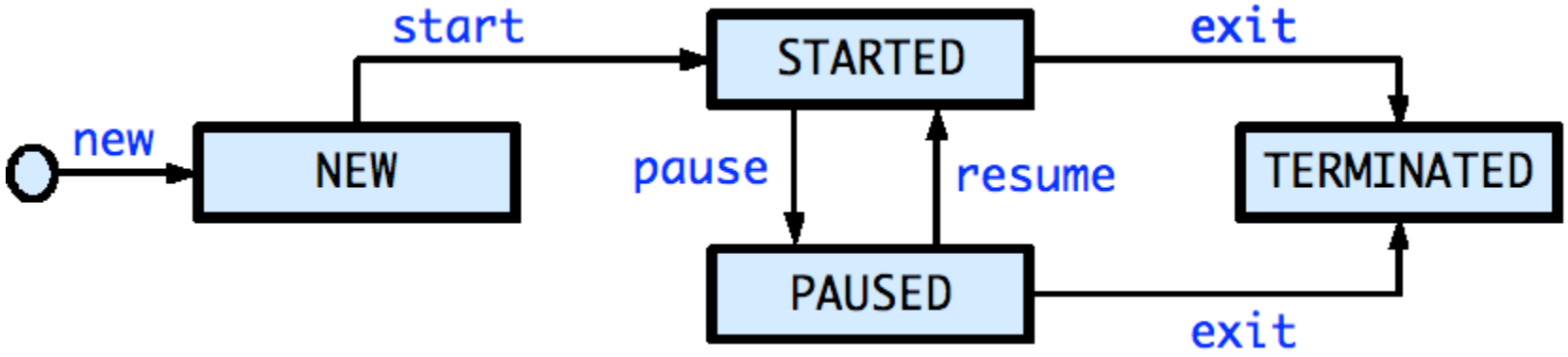
Actor states

- New: Actor has been created
  - e.g., email account has been created
- Started: Actor can receive and process messages
  - **e.g., email account has been activated**
- Terminated: Actor will no longer processes messages
  - **e.g., termination of email account after graduation**

Synchronous Reply using Async-Await

```
1. class SynchronousReplyActor1 extends Actor {
2. void process(Message msg) {
3.     if (msg instanceof Ping) {
4.         finish {
5.             DataDrivenFuture<T> ddf = new DataDrivenFuture<T>();
6.             otherActor.send(ddf);
7.             async await(ddf) {
8.                 T synchronousReply = ddf.get();
9.                 // do some processing with synchronous reply
10.            }
11.        }
12.    } else if (msg instanceof ...) { ... } } }
```


Actors: pause and resume (Recap)



- PAUSED state: actor will not process *subsequent* messages until it is resumed
  - Pausing an actor does not block current process() call
- Pause an actor before returning from message processing body with escaping asyncs
- Resume actor when it is safe to process subsequent messages
  - Messages can accumulate in mailbox when actor is in PAUSED state (analogous to NEW state)

Actors: pause and resume (contd)

- pause() operation:
  - Is a non-blocking operation, i.e. allows the next statement to be executed.
  - Calling pause() when the actor is already paused is a no-op.
  - Once paused, the state of the actor changes and it will no longer process messages sent (i.e. call process(message)) to it until it is resumed.
- resume() operation:
  - Is a non-blocking operation.
  - Calling resume() when the actor is not paused is an error, the HJ runtime will throw a runtime exception.
  - Moves the actor back to the STARTED state
    - the actor runtime spawns a new asynchronous thread to start processing messages from its mailbox.

COMP 322, Spring 2013 (V. Sarkar)

Synchronous Reply using Pause/Resume

```
1.  class SynchronousReplyActor2 extends Actor {
2.    void process(Message msg) {
3.      if (msg instanceof Ping) {
4.        DataDrivenFuture<T> ddf = new DataDrivenFuture<T>();
5.        otherActor.send(ddf);
6.        pause(); // the actor doesn't process subsequent messages
7.        async await(ddf) { // this async processes synchronous reply
8.           T synchronousReply = ddf.get();
9.           // do some processing with synchronous reply
10.          resume(); // allow actor to process next message in mailbox
11.       }
12.     } else if (msg instanceof ...) { ... } } }
```


Worksheet #22: Linearizability of method calls on a concurrent object

Is this a linearizable execution for a FIFO queue, q?

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.

Linearizability of Concurrent Objects (Summary)

Concurrent object

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

Linearizability

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

One Possible Attempt to Implement a Concurrent Queue

```
1.  // Assume that no. of enq() operations is < Integer.MAX_VALUE
2.  class Queue1 {
3.    AtomicInteger head = new AtomicInteger(0);
4.    AtomicInteger tail = new AtomicInteger(0);
5.    Object[] items = new Object[Integer.MAX_VALUE];
6.    public void enq(Object x) {
7.      int slot = tail.getAndIncrement(); // isolated(tail) ...
8.     items[slot] = x;
9.   } // enq
10.   public Object deq() throws EmptyException {
11.     int slot = head.getAndIncrement(); // isolated(head) ...
12.     Object value = items[slot];
13.     if (value == null) throw new EmptyException();
14.     return value;
15.   } // deq
16. } // Queue1

17. // Client code
18. finish {
19.   Queue1 q = new Queue1();
20.   async q.enq(new Integer(1));
21.   q.enq(newInteger(2));
22.   Integer x = (Integer) q.deq();
23. }
```

Worksheet #23: Is there a possible execution for which deq() results in an EmptyException? If so, that is a non-linearizable execution.

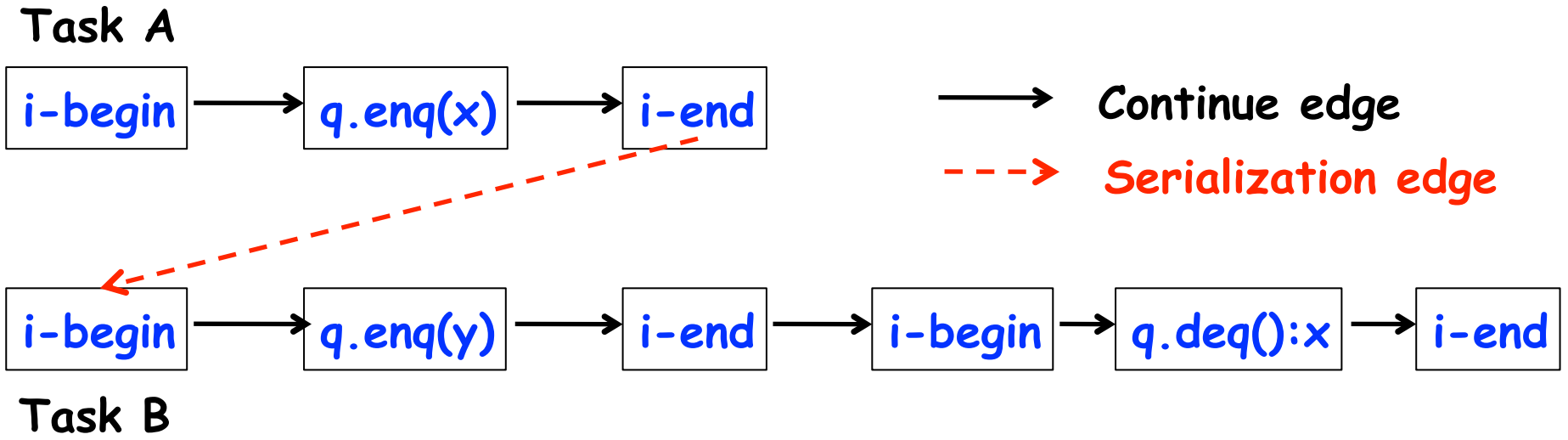
Example 4: execution of a monitor-based implementation of FIFO queue q (Recap)

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()`

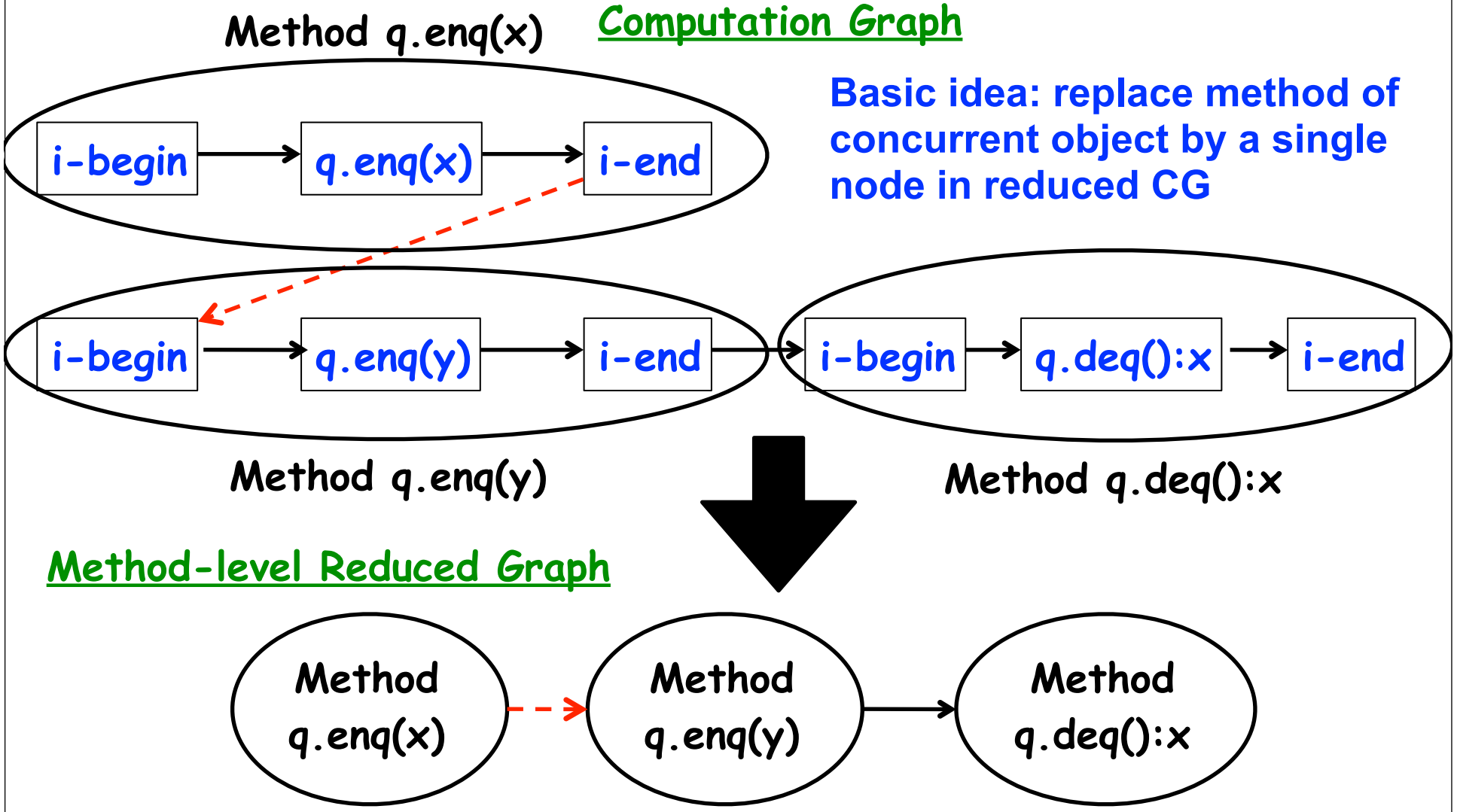
Yes! Equivalent to "q.enq(x) ; q.enq(y) ; q.deq():x"

Computation Graph for previous execution (Example 4)



Monitor-based execution encloses each method call in an isolated statement, demarcated by isolated-begin (i-begin) and isolated-end (i-end) nodes

Creating a Reduced Computation Graph to model Instantaneous Execution of Methods in a Concurrent Object



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 the previous example.
- 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.
  - —We cannot use a cycle in the reduced CG as evidence of non-linearizability

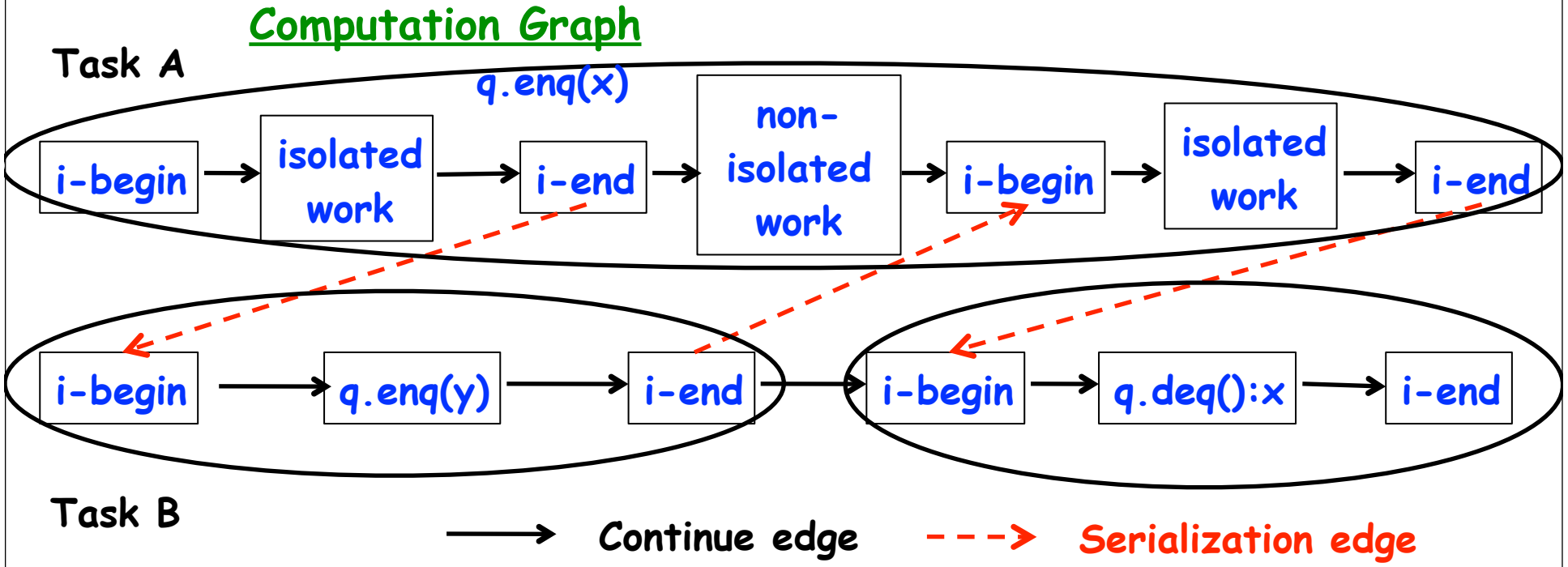
Example 5: Example execution of method calls on a concurrent FIFO queue q (Recap)

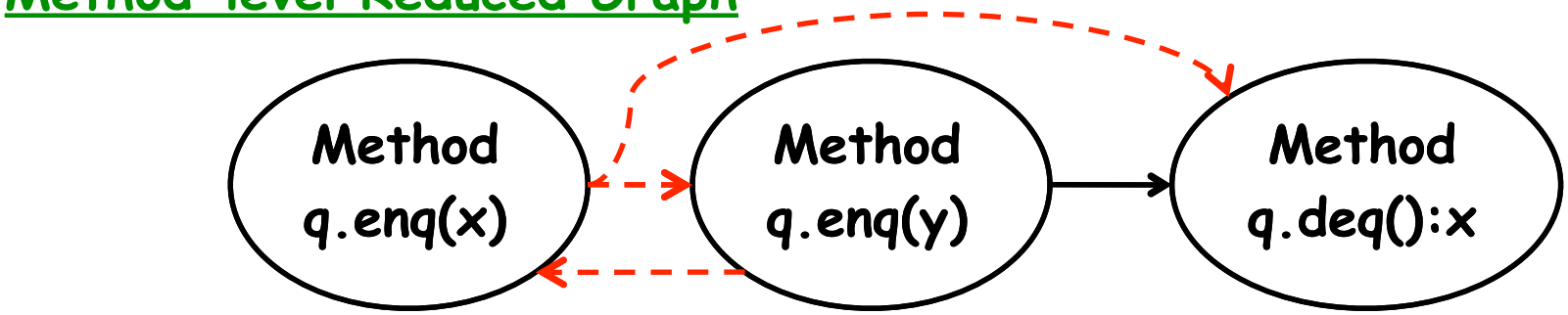
Is this a linearizable execution?

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"

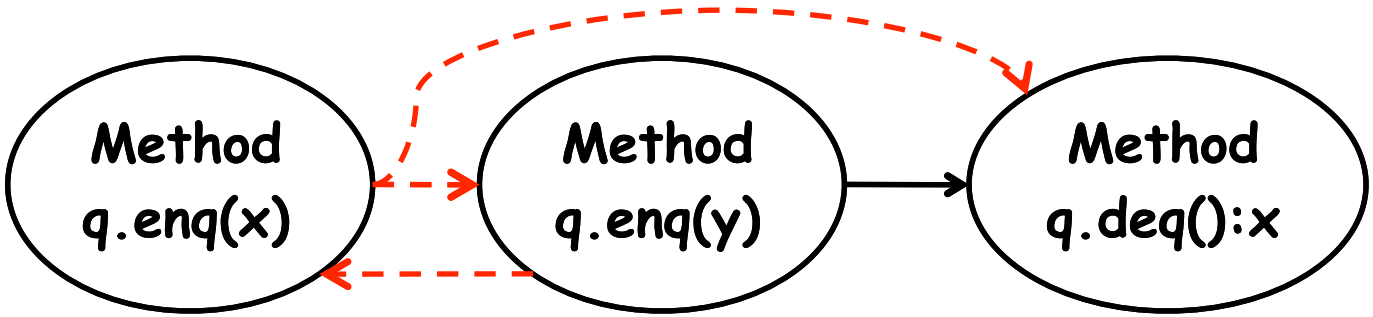
Computation Graph for previous execution (Example 5)





Reduced Computation Graph for previous execution (Example 5)

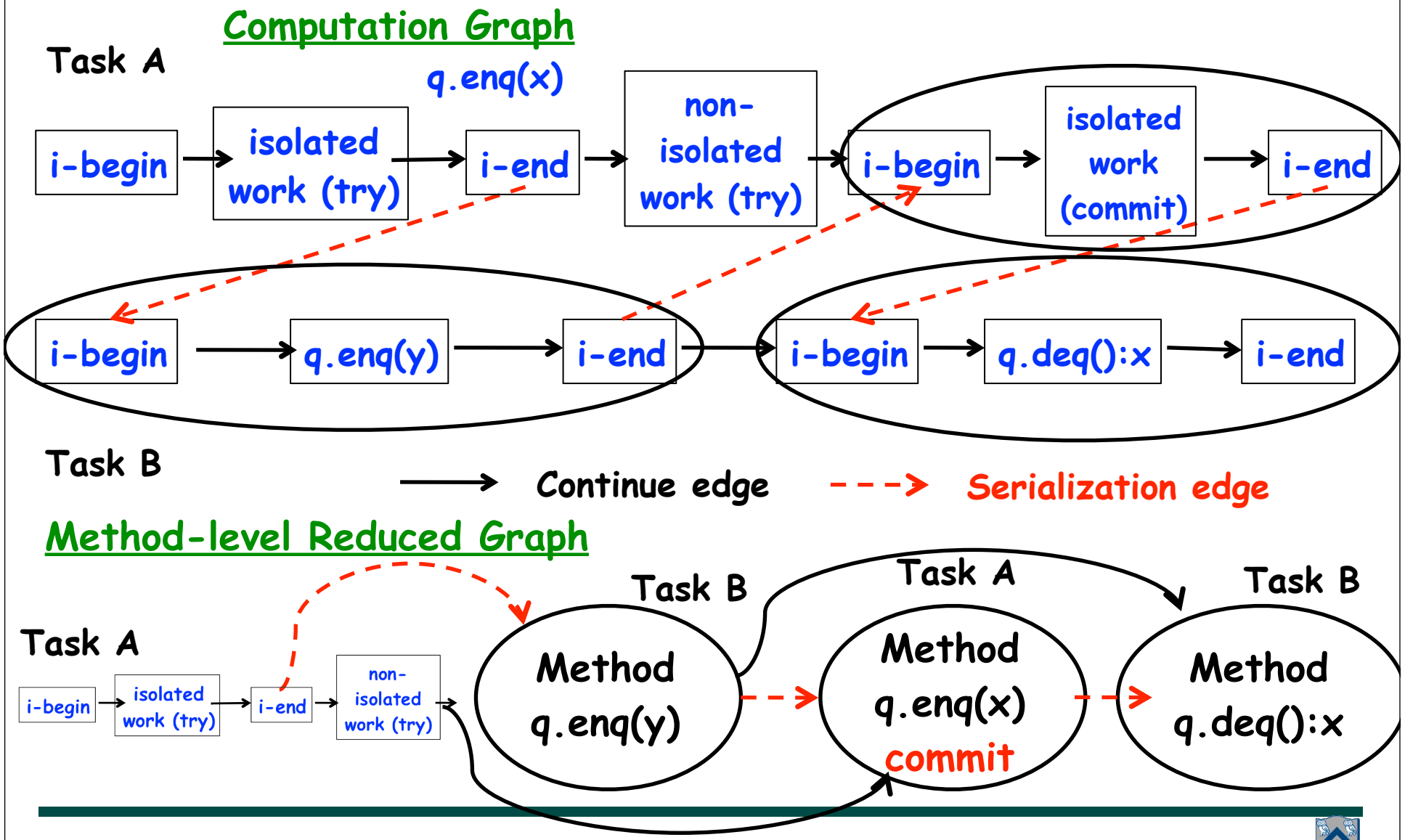
- Example of linearizable execution graph for which reduced method-level graph is cyclic



- Approach to make cycle test more precise for linearizability
  - Decompose concurrent object method into a sequence of "try" steps followed by a "commit" step
    - "try" steps are usually implemented as a loop (this notion of "try" is unrelated to Java's try-catch statements)
  - Assume that each "commit" step's execution does not use any input from any prior "try" step

➔ Reduced graph can just reduce the "commit" step to a single node instead of reducing the entire method to a single node

Computation Graph for Example 5 decomposed into try & commit portions



Motivation for try-commit pattern

- "Nonblocking" synchronization

— Pro: Resilient to failure or delay of any thread attempting synchronization

— Con: "spin loop" may tie up a worker indefinitely

- *Try-in-a-loop* pattern for optimistic synchronization

```
LOOP {
    1) Set-up (local operation invisible to other threads)
    2) Instantaneous effect e.g., CompareAndSet
       a) If successful break out of loop
       b) If unsuccessful continue loop
 }
 3) (OPTIONAL) Clean-up if needed (can be done by any task)
```


Example of non-blocking synchronization: implementing AtomicInteger.getAndAdd() using compareAndSet()

```
      /** Atomically adds delta to the current value.
1.     *
2.     * @param delta the value to add
3.     * @return the previous value
4.     */
5.    public final int getAndAdd(int delta) {
6.        for (;;) { // try
7.            int current = get();
8.            int next = current + delta;
9.            if (compareAndSet(current, next))
10.                // commit
11.                return current;
12.        }
13.    }
```

- **Source: http://gee.cs.oswego.edu/cgi-bin/viewcvs.cgi/jsr166/src/main/java/util/concurrent/atomic/AtomicInteger.java**

Worksheet #23: Linearizability of method calls on a concurrent object

Name 1: _____ Name 2: _____

Can you show an execution for which deq() results in an EmptyException in line 22 below? If so, that is a non-linearizable execution.

One Possible Attempt to Implement a Concurrent Queue

```
1.  // Assume that no. of enq() operations is < Integer.MAX_VALUE
2.  class Queue1 {
3.    AtomicInteger head = new AtomicInteger(0);
4.    AtomicInteger tail = new AtomicInteger(0);
5.    Object[] items = new Object[Integer.MAX_VALUE];
6.    public void enq(Object x) {
7.      int slot = tail.getAndIncrement(); // isolated(tail) ...
8.     items[slot] = x;
9.   } // enq
10.   public Object deq() throws EmptyException {
11.     int slot = head.getAndIncrement(); // isolated(head) ...
12.     Object value = items[slot];
13.     if (value == null) throw new EmptyException();
14.     return value;
15.   } // deq
16. } // Queue1

17. // Client code
18. finish {
19.   Queue1 q = new Queue1();
20.   async q.enq(new Integer(1));
21.   q.enq(newInteger(2));
22.   Integer x = (Integer) q.deq();
23. }
```