COMP 322: Fundamentals of Parallel Programming

Lecture 24: Monitors, Java Concurrent Collections, Linearizability of Concurrent Objects

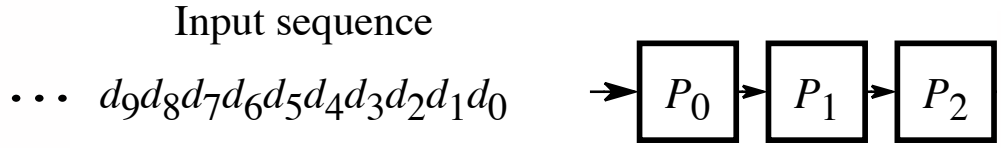
Vivek Sarkar
Department of Computer Science, Rice University
vsarkar@rice.edu

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

Solution to Worksheet #23: Ideal Parallelism in Actor Pipeline

Consider a three-stage pipeline of actors set up so that P0.nextStage = P1, P1.nextStage = P2, and P2.nextStage = null. The process() method for each actor is shown below. Assume that 100 non-null messages are sent to actor P0 after all three actors are started, followed by a null message. What will the total WORK and CPL be for this execution? Recall that each actor has a sequential thread.

Solution: WORK = 300, CPL = 102




```
1.          protected void process(final Object msg) {
2.              if (msg == null) {
3.                  exit();
4.              } else {
5.                  doWork(1); // unit work
6.              }
7.              if (nextStage != null) {
8.                  nextStage.send(msg);
9.              }
10.         }
```


Monitors --- an object-oriented approach to isolation

- A monitor is an object containing
  - some local variables (private data)
  - methods that operate on local data (monitor regions)
- Only one task can be active in a monitor at a time, executing some monitor region
  - **Analogous to a critical section**
  - **As if each public method is an isolated construct**
- Monitors can also be used for
  - Mutual exclusion
  - Cooperation
  - In Operating Systems, a monitor is viewed as a high-level view of semaphores

Monitors – a Diagrammatic summary

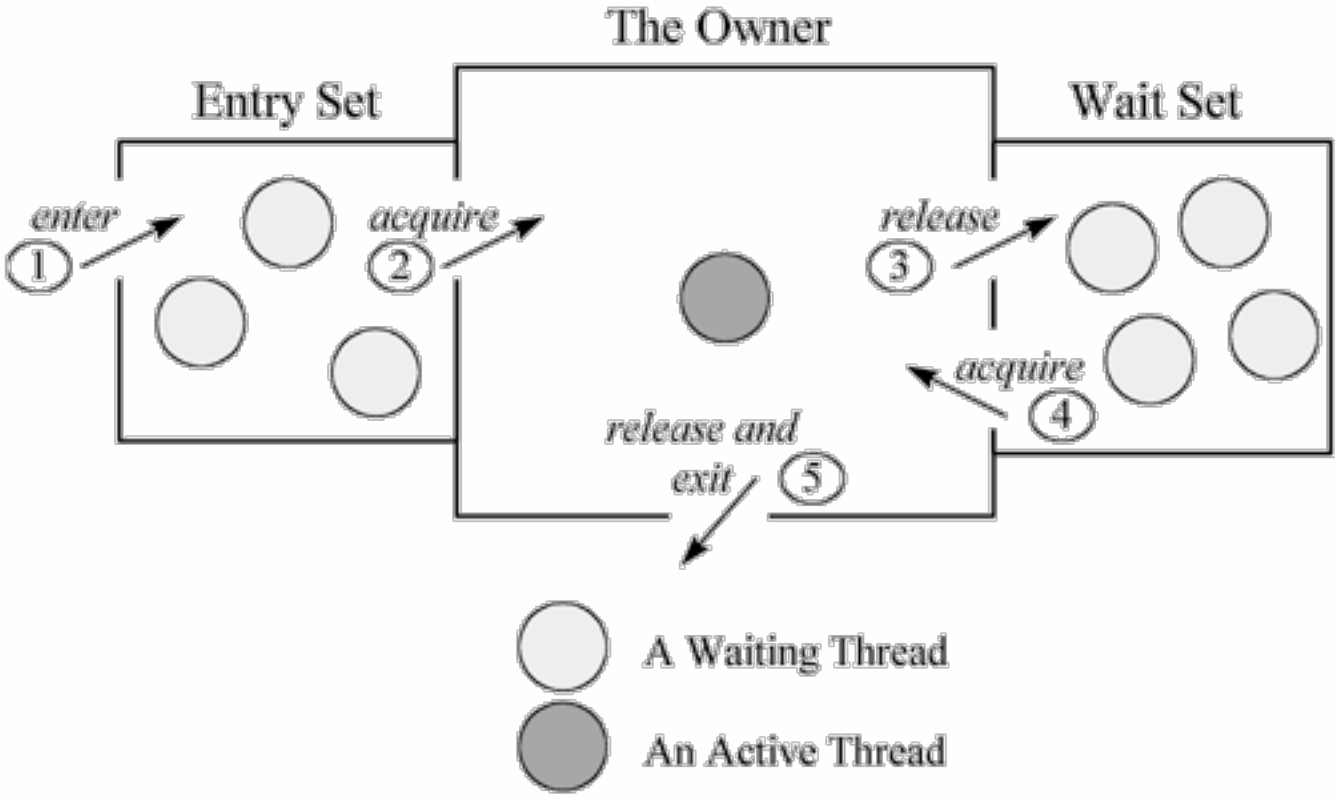


Figure 20-1. A Java monitor.

Figure source: http://www.artima.com/insidejvm/ed2/images/fig20-1.gif

Converting Standard Java Libraries to Monitors

Different approaches:

1. Restrict access to a single task ➔ no modification needed
2. Ensure that each call to a public method is isolated ➔ excessive serialization
3. Use specialized implementations that minimize serialization across public methods ➔ Java Concurrent Collections

- We will focus on three java.util.concurrent classes that can be used freely in HJ programs, in addition to Java Atomic Variables
  - ConcurrentHashMap, ConcurrentLinkedQueue, CopyOnWriteArraySet
- Other j.u.c. classes can be used in standard Java, but not in HJ because they may perform blocking operations
  - ArrayBlockingQueue, CountDownLatch, CyclicBarrier, DelayQueue, Exchanger, FutureTask, LinkedBlockingQueue, Phaser PriorityBlockingQueue, Semaphore, SynchronousQueue

Concurrent Objects

- A concurrent object is an object that can correctly handle methods invoked in parallel by different tasks or threads
  - Optimized variant of monitors (permits parallel method invocations)
  - 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

java.util.concurrent library

- Atomic variables
  - Efficient implementations of special-case patterns of isolated statements
- Concurrent Collections:
  - Queues, blocking queues, concurrent hash map, …
  - Data structures designed for concurrent environments
- Executors, Thread pools and Futures
  - Execution frameworks for asynchronous tasking
- Locks and Conditions
  - More flexible synchronization control
  - Read/write locks
- Synchronizers: Semaphore, Latch, Barrier, Exchanger, Phaser
  - Tools for thread coordination
- WARNING: only a small subset of the full java.util.concurrent library can safely be used in HJ programs
  - Atomic variables and some concurrent collections are part of the safe subset
  - We will study the full library later this semester as part of Java Concurrency

The Java Map Interface

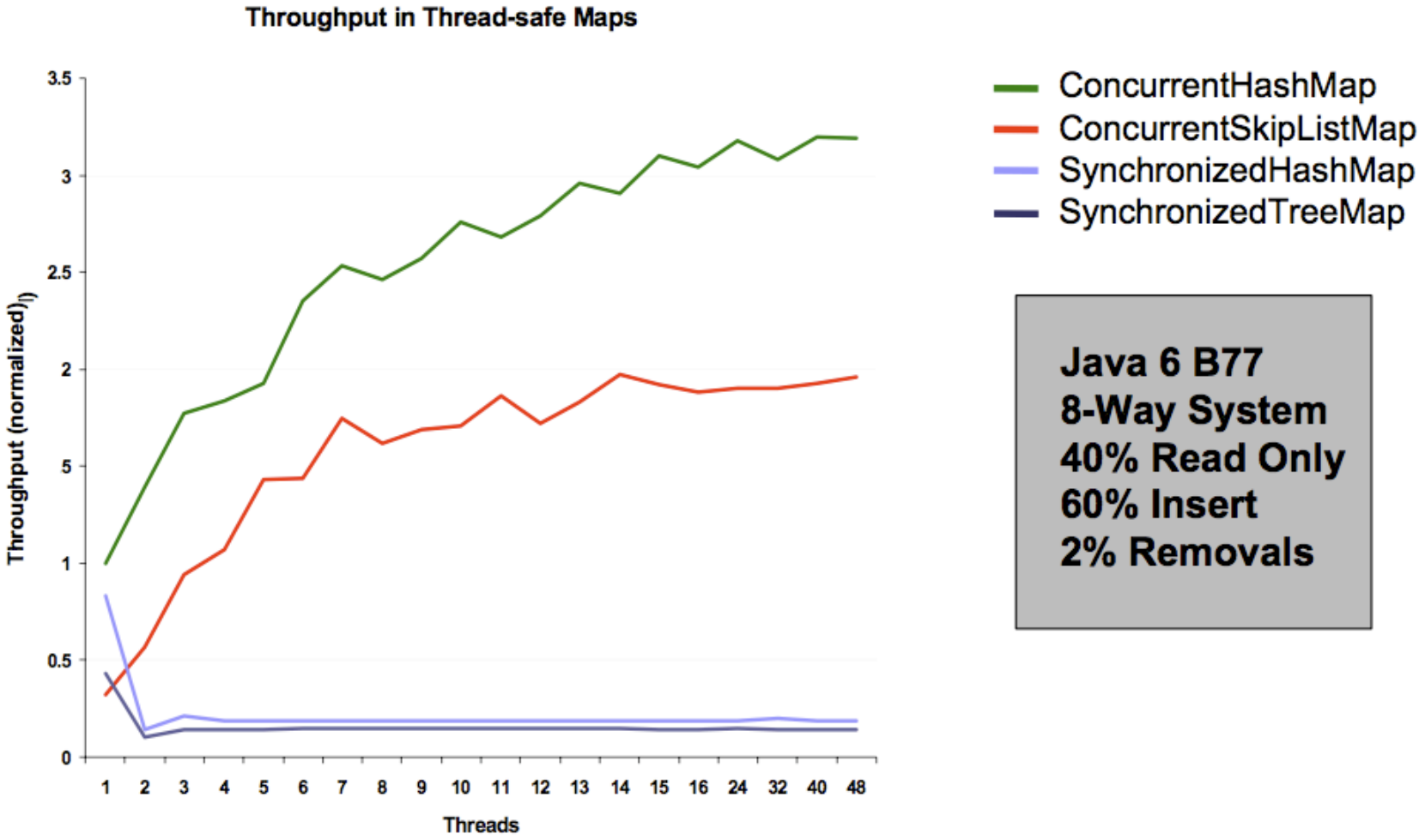
- Map describes a type that stores a collection of key-value pairs
- A Map associates a key with a value
- The keys must be unique
  - the values need not be unique
- Useful for implementing software caches (where a program stores key-value maps obtained from an external source such as a database), dictionaries, sparse arrays, …

- A Map is often implemented with a hash table (HashMap)
- Hash tables attempt to provide constant-time access to objects based on a key (String or Integer)
  - key could be your Student ID, your telephone number, social security number, account number, …
- The direct access is made possible by converting the key to an array index using a hash function that returns values in the range 0 … ARRAY_SIZE-1, typically by using a (mod ARRAY_SIZE) operation

java.util.concurrent.concurrentHashMap

- Implements ConcurrentMap sub-interface of Map
- Allows read (traversal) and write (update) operations to overlap with each other
- Some operations are atomic with respect to each other e.g.,
  - get(), put(), putIfAbsent(), remove()
- Aggregate operations may not be viewed atomically by other operations e.g.,
  - putAll(), clear()
- Expected degree of parallelism can be specified in ConcurrentHashMap constructor
  - ConcurrentHashMap(initialCapacity, loadFactor, concurrencyLevel)
  - A larger value of concurrencyLevel results in less serialization, but a larger space overhead for storing the ConcurrentHashMap

Concurrent Collection Performance



Example usage of ConcurrentHashMap in org.mirrorfinder.model.BaseDirectory

```
1  public abstract class BaseDirectory extends BaseItem implements Directory {
2    Map files = new ConcurrentHashMap();
3    . . .
4    public Map getFiles() {
5      return files;
6    }
7    public boolean has(File item) {
8      return getFiles().containsValue(item);
9    }
10   public Directory add(File file) {
11     String key = file.getName();
12     if (key == null) throw new Error(. . .);
13     getFiles().put(key, file);
14     . . .
15     return this;
16   }
17   public Directory remove(File item) throws NotFoundException {
18     if (has(item)) {
19       getFiles().remove(item.getName());
20       . . .
21     } else throw new NotFoundException("can't remove unrelated item");
22   }
23 }
```

Listing 1: Example usage of ConcurrentHashMap in org.mirrorfinder.model.BaseDirectory [1]

java.util.concurrent.ConcurrentLinkedQueue

- `Queue` interface added to `java.util`
  - interface Queue extends Collection and includes

```
boolean offer(E x); // same as add() in Collection
E poll(); // remove head of queue if non-empty
E remove(o) throws NoSuchElementException;
E peek(); // examine head of queue without removing it
```

- Non-blocking operations
  - —offer() returns `false` when full
  - —poll() returns `null` when empty
- Fast thread-safe non-blocking implementation of Queue interface: `ConcurrentLinkedQueue`

Example usage of ConcurrentLinkedQueue in org.apache.catalina.tribes.io.BufferPool15Impl

```
1  class BufferPool15Impl implements BufferPool.BufferPoolAPI {
2    protected int maxSize;
3    protected AtomicInteger size = new AtomicInteger(0);
4    protected ConcurrentLinkedQueue queue = new ConcurrentLinkedQueue();
5    . . .
6    public XByteBuffer getBuffer(int minSize, boolean discard) {
7      XByteBuffer buffer = (XByteBuffer) queue.poll();
8      if ( buffer != null ) size.addAndGet(-buffer.getCapacity());
9      if ( buffer == null ) buffer = new XByteBuffer(minSize,discard);
10     else if ( buffer.getCapacity() <= minSize ) buffer.expand(minSize);
11     . . .
12     return buffer;
13   }
14   public void returnBuffer(XByteBuffer buffer) {
15     if ( (size.get() + buffer.getCapacity()) <= maxSize ) {
16       size.addAndGet(buffer.getCapacity());
17       queue.offer(buffer);
18     }
19   }
20 }
```

Listing 2: Example usage of ConcurrentLinkedQueue in org.apache.catalina.tribes.io.BufferPool15Impl

java.util.concurrent.CopyOnWriteArraySet

- Set implementation optimized for case when sets are not large, and read operations dominate update operations in frequency
- This is because update operations such as add() and remove() involve making copies of the array
  - Functional approach to mutation
- Iterators can traverse array "snapshots" efficiently without worrying about changes during the traversal.

Example usage of CopyOnWriteArraySet in org.norther.tammi.spray.freemarker.DefaultTemplateLoader

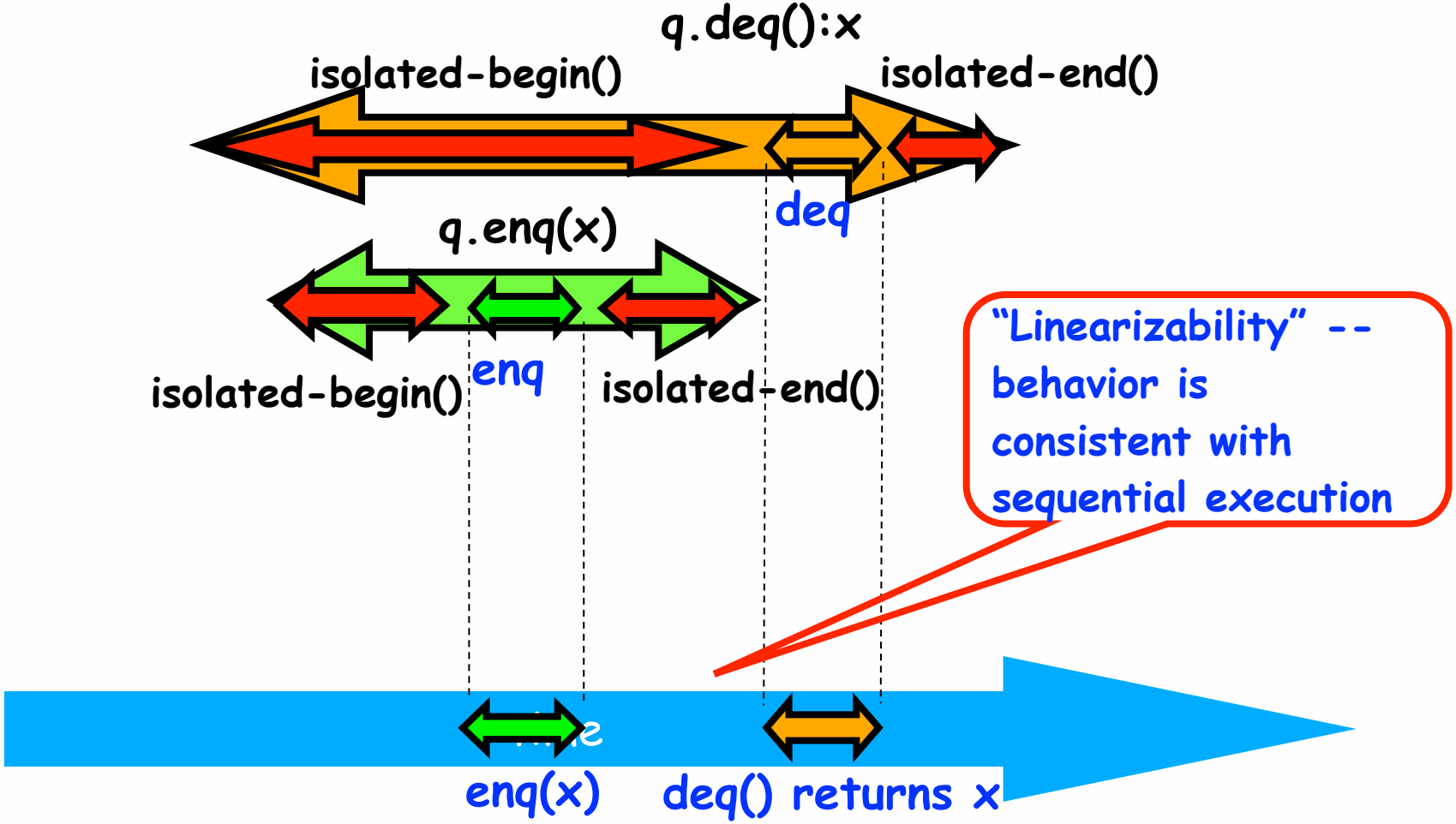
```
1  public class DefaultTemplateLoader implements TemplateLoader, Serializable
2  {
3    private Set resolvers = new CopyOnWriteArraySet();
4    public void addResolver(ResourceResolver res)
5    {
6      resolvers.add(res);
7    }
8    public boolean templateExists(String name)
9    {
10       for (Iterator i = resolvers.iterator(); i.hasNext();) {
11         if (((ResourceResolver) i.next()).resourceExists(name)) return true;
12       }
13       return false;
14   }
15   public Object findTemplateSource(String name) throws IOException
16   {
17     for (Iterator i = resolvers.iterator(); i.hasNext();) {
18       CachedResource res = ((ResourceResolver) i.next()).getResource(name);
19       if (res != null) return res;
20     }
21     return null;
22   }
23  }
```

Listing 3: Example usage of CopyOnWriteArraySet in org.norther.tammi.spray.freemarker.DefaultTemplateLoader

Correctness of a Concurrent Object

- 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

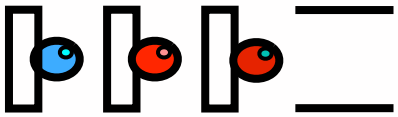


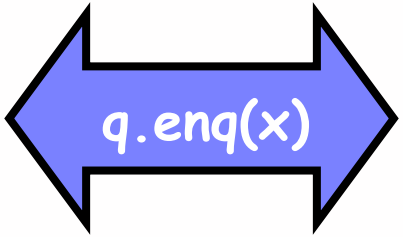
Source: http://www.elsevierdirect.com/companions/9780123705914/Lecture%20Slides/03~Chapter_03.ppt

Informal definition of Linearizability

- 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
- A concurrent object is linearizable if all its executions are linearizable.

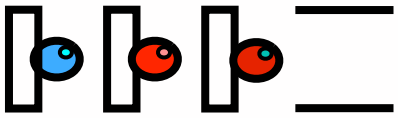
Example 1

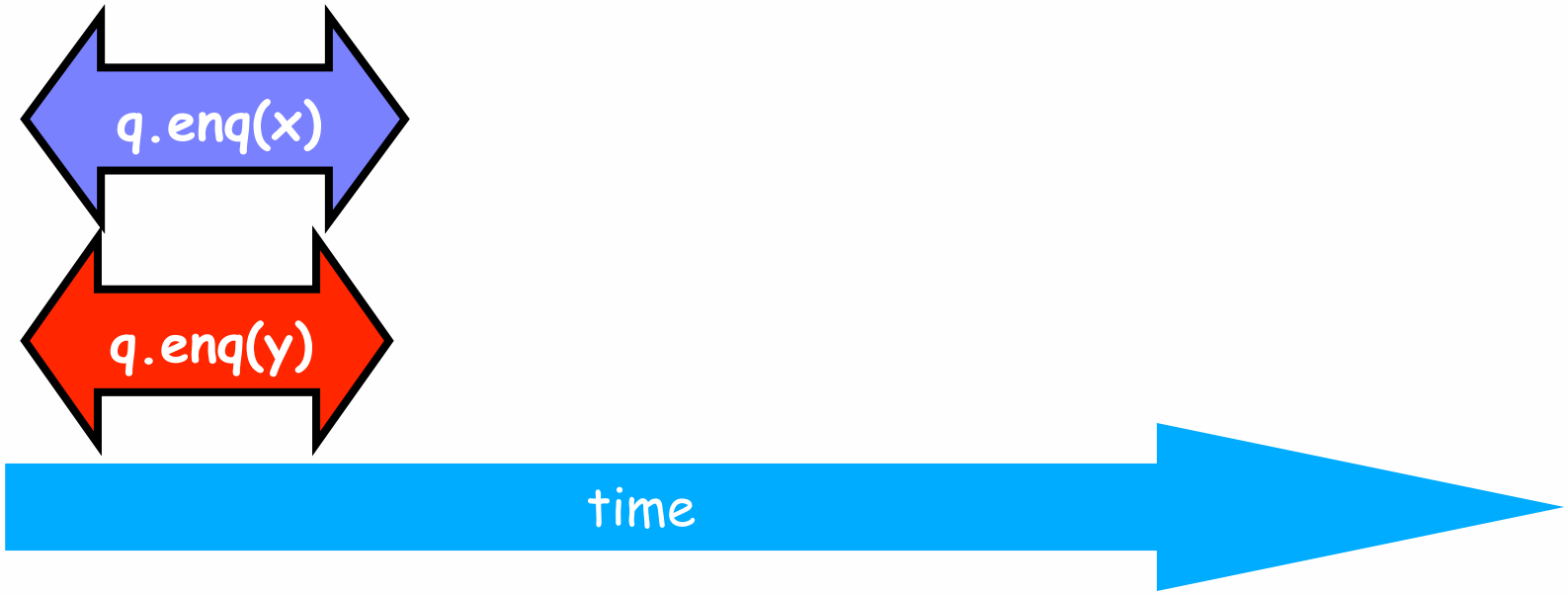




Source: http://www.elsevierdirect.com/companions/9780123705914/Lecture%20Slides/03~Chapter_03.ppt

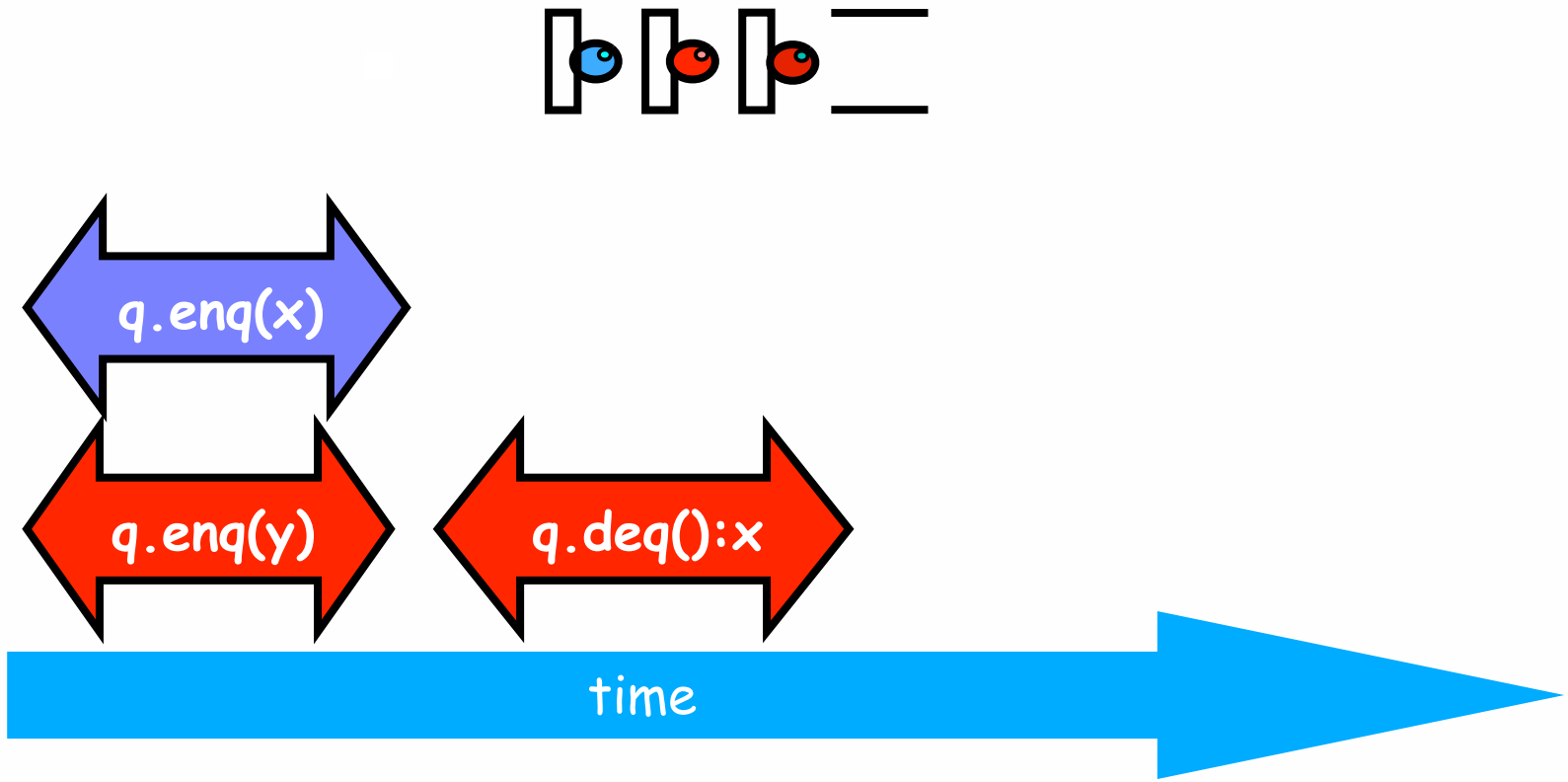
Example 1 (contd)





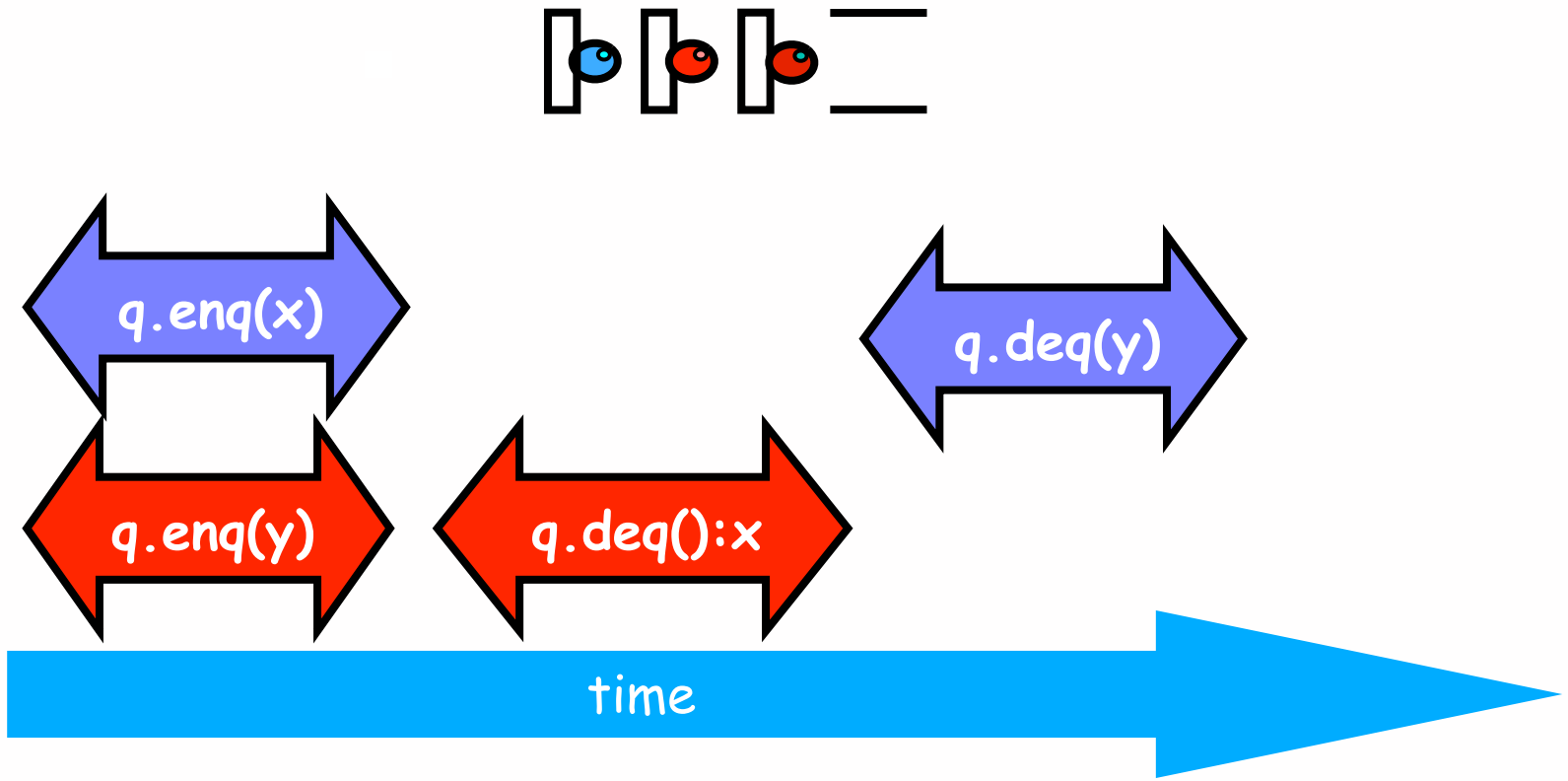
Source: http://www.elsevierdirect.com/companions/9780123705914/Lecture%20Slides/03~Chapter_03.ppt

Example 1 (contd)



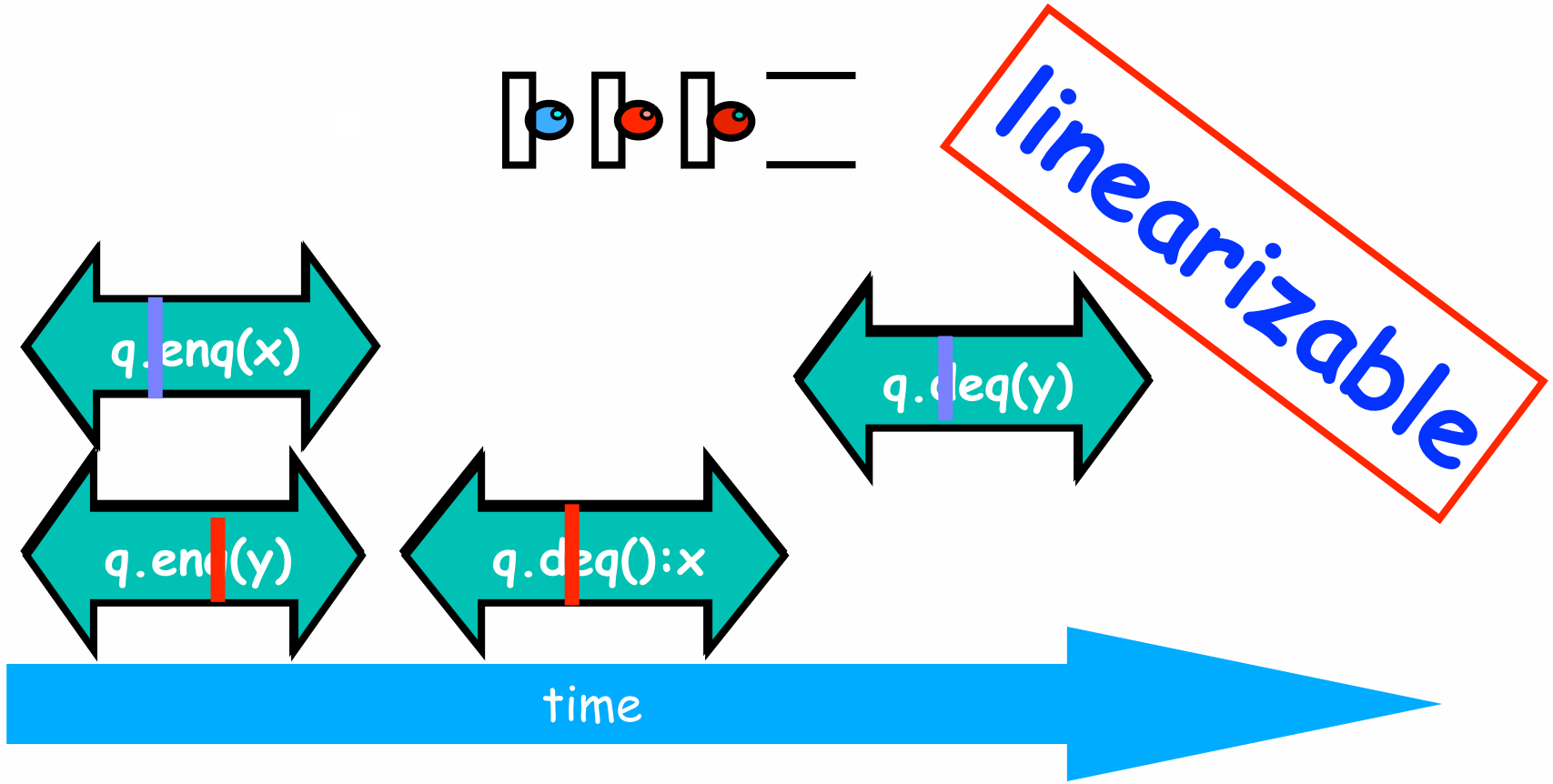
Source: http://www.elsevierdirect.com/companions/9780123705914/Lecture%20Slides/03~Chapter_03.ppt

Example 1 (contd)



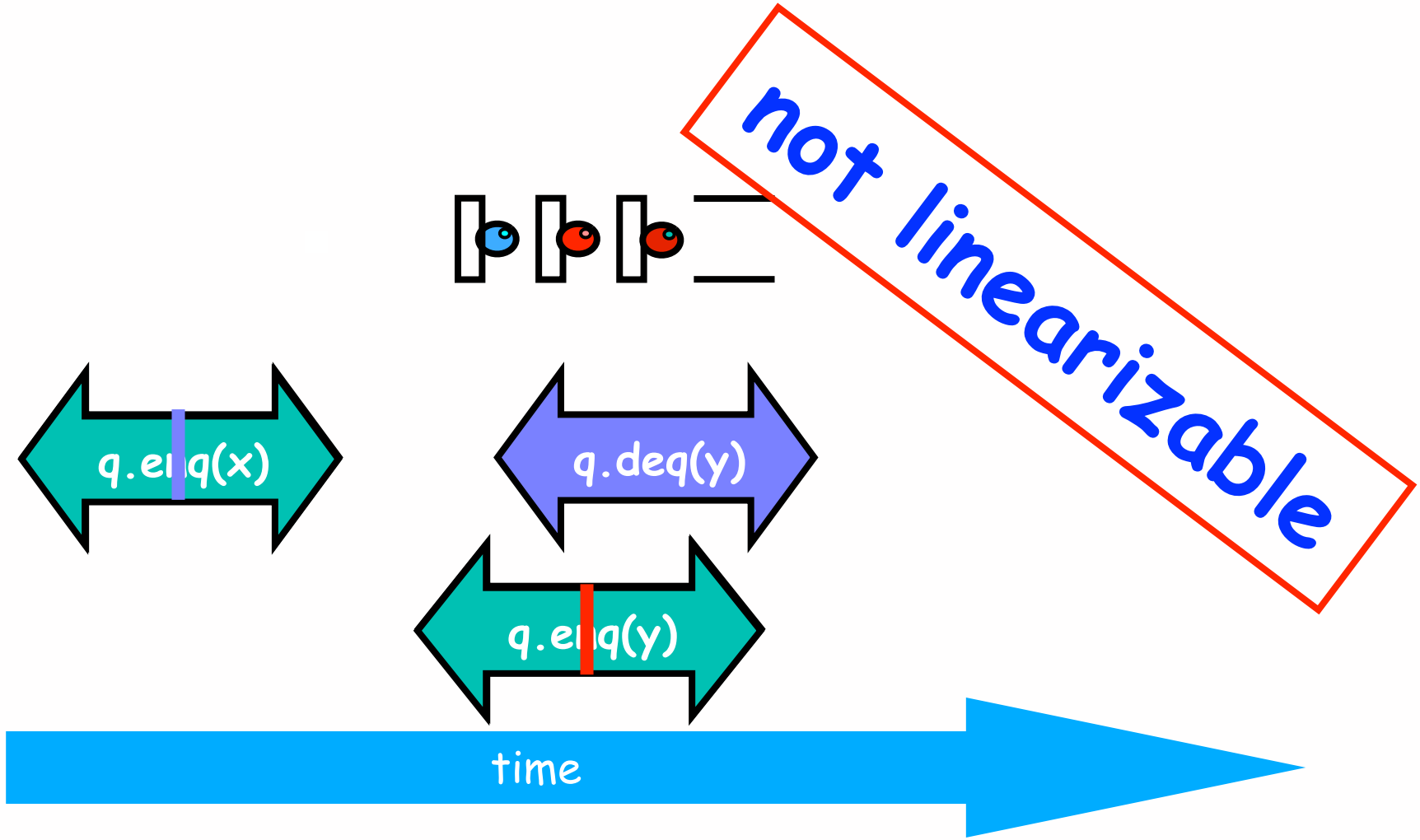
Source: http://www.elsevierdirect.com/companions/9780123705914/Lecture%20Slides/03~Chapter_03.ppt

Example 1 (contd)



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

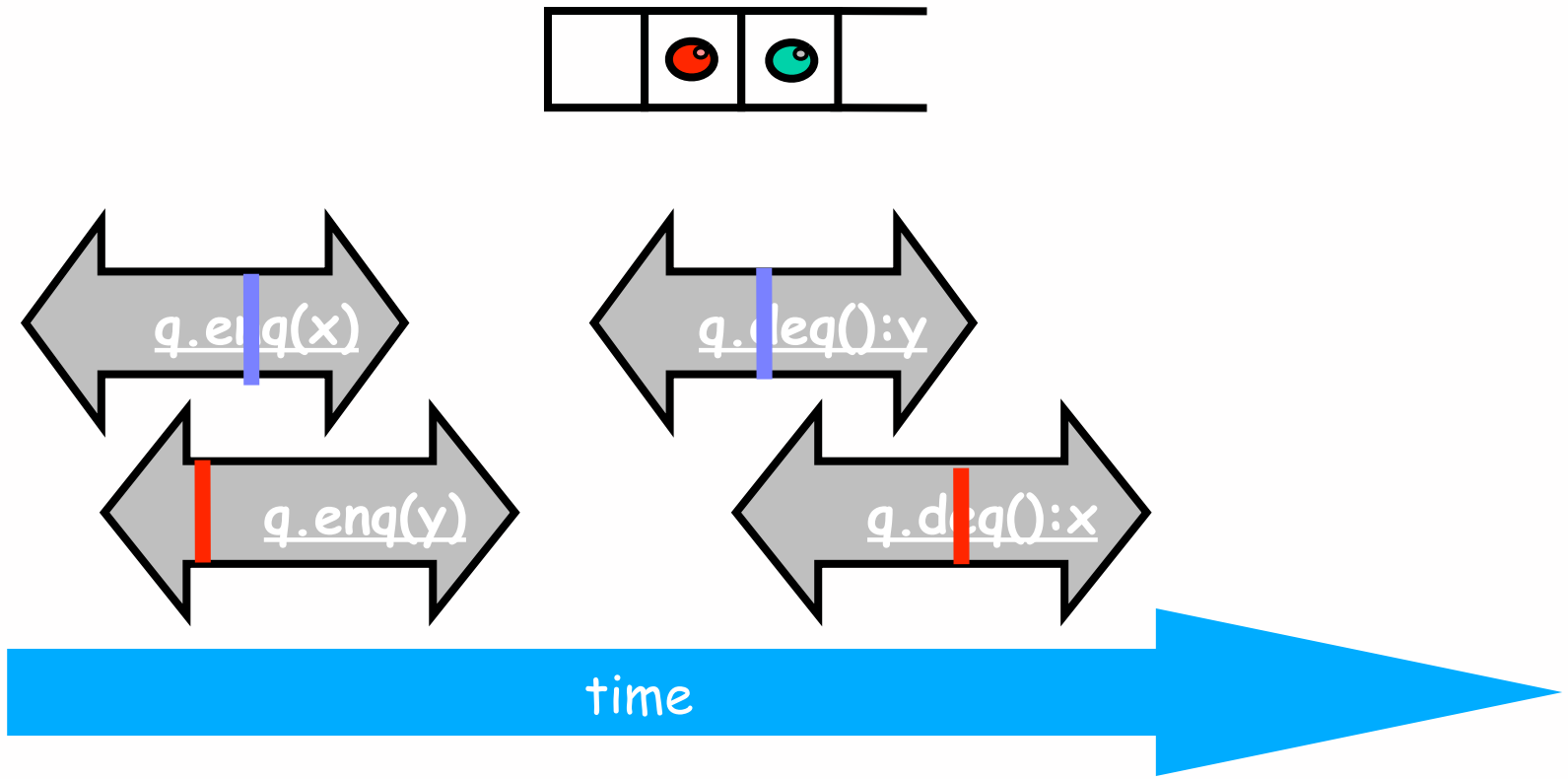
Example 2



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

Example 3

Is this execution linearizable? How many possible linearizations does it have?



Example 4: 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"

Example 5: 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"

Linearizability of Concurrent Objects (Summary)

Concurrent object

- A concurrent object is an object that can correctly handle methods invoked in parallel by 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

Worksheet #24: Linearizability of method calls on a concurrent object

Name: _____ Netid: _____

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()`