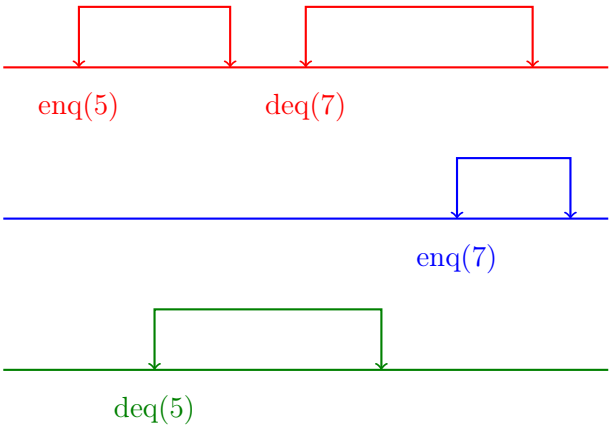
Abstract Data Types, Linearizability

1 Abstract Data Types

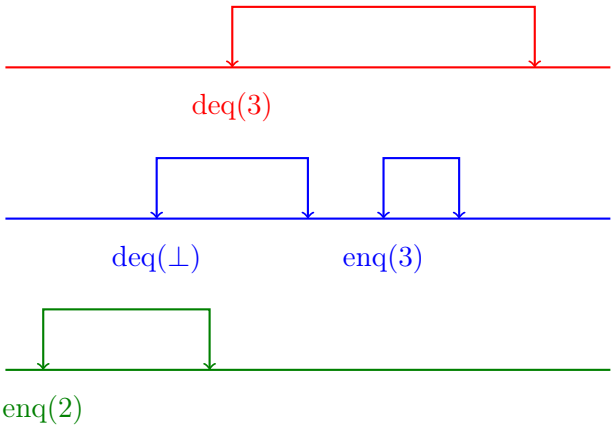
Question 1. Define the abstract data type of Queue (with `enq` and `deq` methods) and Stacks (with `push` and `pop` methods).

Note: `deq` or `pop` return an element (added by `enq` or `push`), or a special "empty" value (ex: $\perp$)

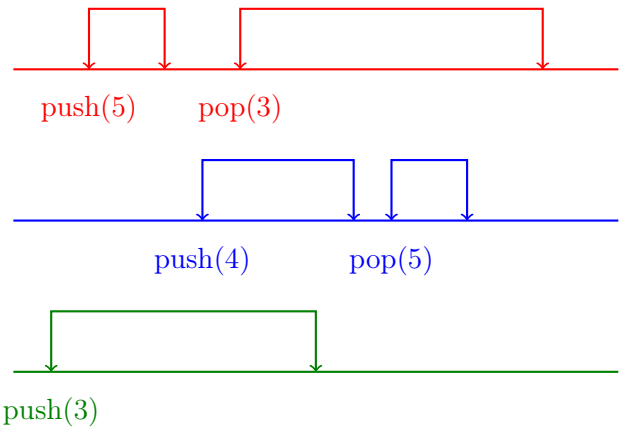
Question 2. Can you determine if the histories of a shared queue and stack provided below are linearizable?



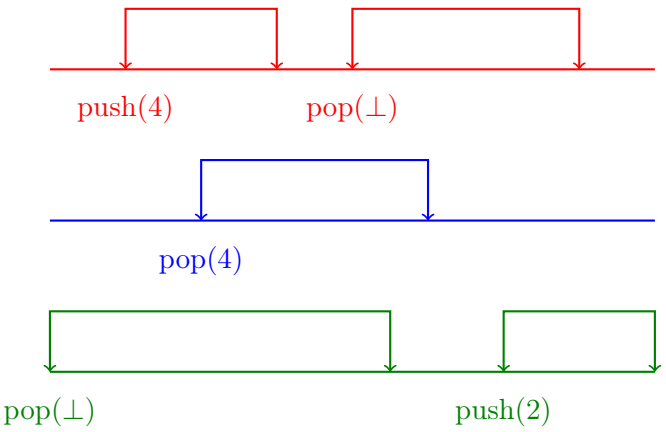
Queue: Execution 1



Queue: Execution 2

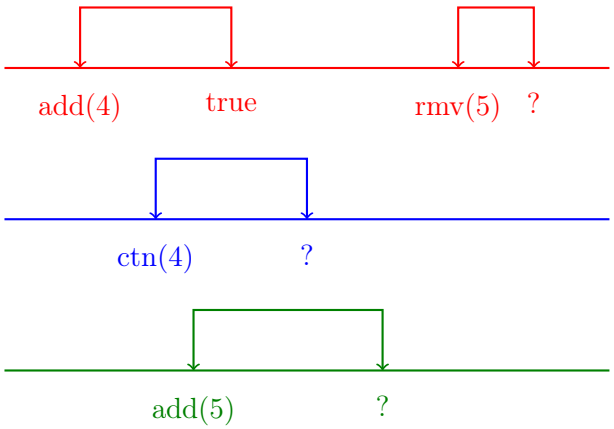


Stack: Execution 1

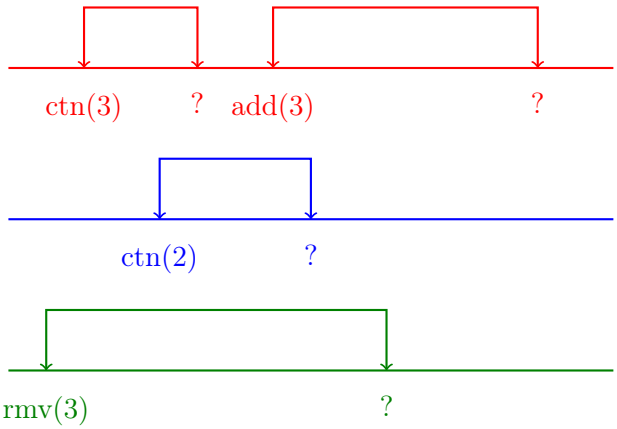


Stack: Execution 2

Question 3. For each history below, what are some possible return values that make the histories linearizable, and what values make them non-linearizable?



Set: Execution 1



Set: Execution 2

2 Multi-Threading and Linearizability

Here is the Java `Thread` lambda syntax to specify the code that a thread should execute, start it, and wait for it to finish:

```
Counter c = new Counter(); // an object that all threads can access

Thread t1 = new Thread(() -> {
    // Code the thread should execute goes here ...
    for (int i = 0; i < 100; i++) {
        c.increment();
    }
});

Thread t2 = new Thread(() -> {
    // Code the thread should execute goes here ...
    for (int i = 100; i < 200; i++) {
        c.increment();
    }
});

t1.start(); // Starts the thread t1
t2.start(); // Starts the thread t2
t1.join(); // Waits for the thread t1 to finish
t2.join(); // Waits for the thread t2 to finish

assert c.read() == 200 : "the counter should be 200" // assertions to check for
    expected behaviors (the program should be run with ``java -ea .. '')
```

Question 4. Implement a counter with a read and increment operation, as if it would be used in a sequential program (without using locks).

a) Run the program given above (with two threads, each performing an increment on the same counter object) multiple times. Do you observe an assertion violation? If not, can you explain why? Can you perform just one increment per thread and still encounter a violation? To increase the likelihood of encountering an assertion violation, you can add a `Thread.sleep` instruction in the increment function.

b) Add a lock to ensure correct behavior. A lock can be used as follows:

```
import java.util.concurrent.locks.*; // importing the standard lock implementations
```

```
Lock lock = new ReentrantLock(); // declaring a lock

lock.lock();
try {
    // Code that should execute when the lock is taken goes here ...
}
finally {
    lock.unlock();
}
```

c) Generate concurrent histories of the counter using the following schema:

```
import java.util.concurrent.atomic.AtomicInteger; // importing a standard counter
    implementation to attach timestamps to call/return events

AtomicInteger time = new AtomicInteger(0); // declaring a counter

// declaring a shared object ...

Thread t1 = new Thread(() -> {
    String log = new String(""); // using a string to record the sequence of
        call/return events
    for (...) { // if we want to do multiple calls in this thread
        log += "t1: call@" + time.getAndIncrement() + " "; // appending a call
            event with the current time
        // calling a method of the shared object goes here ..
        log += "t1: ret@" + time.getAndIncrement() + " "; // appending a return
            event with the current time
    }
    System.out.println(log); // printing the sequence of call/return events in
        this thread (and the times at which they occur)
});

// the same pattern can be used to construct the log of another thread (times are
    obtained using the same time.getAndIncrement() invocations)
```

Question 5. Implement a set data structure based on linked lists, without using locks. Follow the pseudocode given below:

```
add(k):
    Node p, c
    p = H;
    c = p.next;
    while (c.key < k)
        p = c
        c = c.next
    if (c.key = k)
        return false
    else
        n = new Node(k, -)
        n.next = c
        p.next = n
        return true
```

```
remove(k):
    Node p, c
    p = H;
    c = p.next;
    while (c.key < k)
        p = c
        c = c.next
    if (c.key = k)
        p.next = c.next
        return true
    else
        return false
```

```
ctn(k):
    Node p, c
    p = H;
    c = p.next;
    while (c.key < k)
        p = c
        c = c.next
    if (c.key = k)
        return true
    else
        return false
```

a) It is expected that this implementation is not linearizable. Define concurrent programs where after all threads finished:

- Some elements that should be in the set are missing.

- The set does not contain the expected number of elements (checking this requires adding a `size` method that counts the number of elements in the list).

This is a way to detect linearizability violations similar to the way we did it for the shared counter. Try to see if you can find other ways of identifying linearizability violations in the set implementation.

b) Generate histories of the set operations that are not linearizable. You can add `Thread.sleep` instructions to increase their likelihood.

Question 6. Add locks to the set implementation to ensure that it is linearizable.

a) Generate execution histories of the locked version.

b) Demonstrate that these histories are linearizable by identifying the linearization points for each operation.

Question 7. Compare the performance of your locked set implementation with Java's `ConcurrentSkipListSet`. See the code snippet below for usage:

```
import java.util.concurrent.ConcurrentSkipListSet;

public class TimeComparison {
    public static void main(String[] args) {
        ConcurrentSkipListSet<Integer> set = new ConcurrentSkipListSet<>();

        // Insertion
        set.add(1);
        set.add(2);

        // Lookup
        boolean exists = set.contains(1);

        // Removal
        set.remove(2);
    }
}
```

Here is a snippet to record starting/ending time

```
long start = System.currentTimeMillis();
// do something ...
long finish = System.currentTimeMillis();
long timeElapsed = finish - start;
```