Question1

We suppose the elements of queue and stack are integers.

We use lower case letters to represent a list element, and upper case letters to represent a list.

We use the notation E to represent an empty list, and x, L, y to represent the list whose first element is x, the middle being L, and last element is y. Any of the three parts can be omitted.

Queue

Set of states: set of lists

Initial state: E

Set of method names: $\{enq, deq\}$

Set of rules:

$$L \xrightarrow{enq(x)} x, L$$

$$L, x \xrightarrow{deq(x)} L$$

$$E \xrightarrow{deq(\perp)} E$$

Stack

Set of states: set of lists

Initial state: E

Set of method names: $\{push, pop\}$

Set of rules:

$$L \xrightarrow{push(x)} x, L$$

$$x, L \xrightarrow{pop(x)} L$$

$$E \xrightarrow{pop(\perp)} E$$

Question2

Queue Execution 1 is linearizable, there exists a possible trace :

$$enq(5)\ \ deq(5)\ \ enq(7)\ \ deq(7)$$

Queue Execution 2 is not linearizable, since $enq(2)$ precedes all other enq operations, the first deq operation after that must return 2.

Stack Execution 1 is not linearizable, since there are three $push$ operations and $push(5)$ precedes $push(4)$, 4 cannot be at the bottom of stack thus must be returned by either of the two pop operations.

Stack Execution 2 is linearizable, there exists a possible trace :

$$pop(\bot)\ \ push(4)\ \ pop(4)\ \ pop(\bot)\ \ push(2)$$

Question3

Set Execution 1:

To make the history linearizable:

- $cnt(4, true)$
- $add(5, true)$
- $rmv(5, false)$

or

- $cnt(4, false)$
- $add(5, true)$
- $rmv(5, false)$

Other return values make the history non-linearizable.

Set Execution 2:

To make the history linearizable:

- $cnt(3, false)$
- $cnt(2, false)$
- $rmv(3, true)$
- $add(3, true)$

or

- $cnt(3, false)$
- $cnt(2, false)$
- $rmv(3, false)$
- $add(3, true)$

Question4

Implementation of `Counter` without using blocks:

```java
public class Counter {
    public int count = 0;

    public void increment() {
        int temp;
        // begin of crucial part
        temp = count;
        temp = temp + 1;
        count = temp;
        // end of crucial part
    }

    public int read() {
        int temp = count;
        return temp;
    }
}
```

a) Running the program multiple times, an assertion violation can sometimes be observed. There are also times where the assertion succeed.

If the assertion succeed, this is because when the two threads run concurrently, the crucial part in method `increment` happen to never overlap with each other.

If we just perform one increment per thread, this reduces the chance of overlap. We can increase the change by prolonging the execution time of crucial part. For example:

```java
    public void increment() {
        int temp;
        // begin of crucial part
        try {
            Thread.sleep(5000);
        } catch (Exception e) {
            System.out.println("Exception");
        }
        temp = count;
        temp = temp + 1;
        count = temp;
        // end of crucial part
    }
```

b) By adding a lock when executing the crucial part, we ensure that that block of code is not executed simultaneously by two threads. Using such a `Counter` object ensures that the assertion always succeed.

```java
import java.util.concurrent.locks.*;

public class Counter {
    public int count = 0;
    Lock lock = new ReentrantLock();

    public void increment() {
        int temp;
        lock.lock();
        try {
            // begin of crucial part
            temp = count;
            temp = temp + 1;
            count = temp;
            // end of crucial part
        } finally {
            lock.unlock();
        }
    }

    public int read() {
        int temp = count;
        return temp;
    }
}
```

c) One generated concurrent history is stored in `history_counter.txt`.

Question5

a) Ways to detect non-linearizability:

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

```java
    public static void main(String[] args) throws Exception {
        Set s = new Set();

        Thread t1 = new Thread(() -> {
            s.add(0);
            s.add(1);
        });

        Thread t2 = new Thread(() -> {
            s.add(2);
            s.add(3);
        });

        t1.start();
        t2.start();
        t1.join();
        t2.join();

        System.out.println(s.toString());
    }
```

Executing this program can yield $\{0, 1\}$ or $\{0, 3\}$ or $\{2, 3\}$ instead of $\{0, 1, 2, 3\}$.

- The set does not contain the expected number of elements:

```
    public static void main(String[] args) throws Exception {
        Set s = new Set();

        Thread t1 = new Thread(() -> {
            for (int i = 0; i < 2; ++i) {
                s.add(i);
            }
        });

        Thread t2 = new Thread(() -> {
            for (int i = 2; i < 4; ++i) {
                s.add(i);
            }
        });

        t1.start();
        t2.start();
        t1.join();
        t2.join();

        System.out.println(s.size());
    }
```

Executing this program prints a size less than 4.

b) One generated concurrent history is stored in `history_unlocked_set.txt`.

Question6

We lock lock at the begining of methods *add* and *rmv*, and unlock the lock at the end of these two methods.

a) One generated concurrent history is stored in `history_locked_set.txt`. We put 4 operation in order to simplify the history.

```
t2: call@0 t2: call@2 t2: call@3 t2: call@4
t1: call@1 t1: call@5 t1: call@6 t1: call@7
```

b) The order of linearization points:

$$add(0, true)\ \ add(1, true)\ \ add(2, true)\ \ add(3, true)$$

Question7

Here is one output of the experiment on adding 10000 elements, querying 10000 times and removing 10000 elements in each thread:

```
Time elapsed using our implementation: 296ms
Time elapsed using java's implementation: 29ms
```

Java's implementation of locked set is 10 times faster then our implementation.