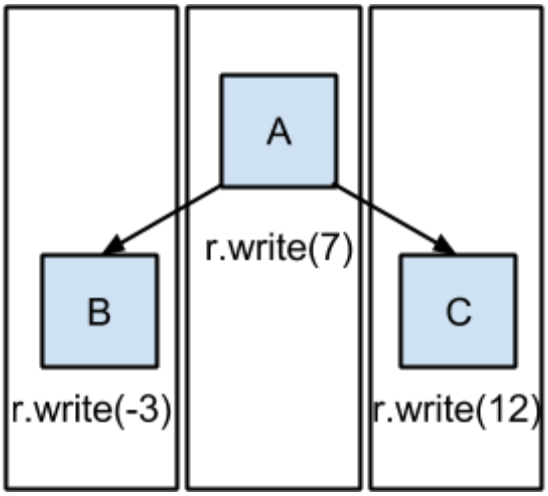
**Exercise 21.** Explain why quiescent consistency is compositional.

Entendendo que o principio Composicional é de que se as varias partes (objetos do sistema) satisfazem uma condição, então o todo (o sistema) irá satisfazer.




Sendo A composto por B e C que são quiescent consistency qualquer período que não houve chamadas de metodos pendentes (quiescent consistency) em A será também em B e C já que os mesmos serão acessados através de A.

**Exercise 23.** Give an example of an execution that is quiescently consistent but not sequentially consistent, and another that is sequentially consistent but not quiescently consistent.

**Consideracões:** A ***consistência de repouso (quiescently consistent)*** descreve uma ordem clara de comportamentos do programa em todos os threads. É necessário um período de repouso entre duas chamadas de método. Pode haver sobreposições, desde que haja um período de repouso entre duas chamadas de método.
A ***consistência sequencial (sequentially consistent)*** permite sobreposições, mas requer que as chamadas de método possam ser organizadas corretamente mantendo a ordem comum das chamadas de método em cada thread. As chamadas em diferentes segmentos podem ser reorganizadas, como desejar, independentemente de quando eles começam e quando terminam.

Consistência Quiescently e não Consistência Sequencial

A ----------r.read(7)------r.read(7)------------------------------

B ----r.write(7)---r.write(-3)---r.read(-3)-------------------------

Consistência Sequencial e não Consistência Quiescently

A - -q.enq(7)--------------q.deq(-3)-----------------------------

B ---------------q.enq(-3)-------------q.deq(7)------------------

**Exercise 24.** For each of the histories shown in Figs. 3.13 and 3.14, are they quiescently consistent? Sequentially consistent? Linearizable? Justify your answer.

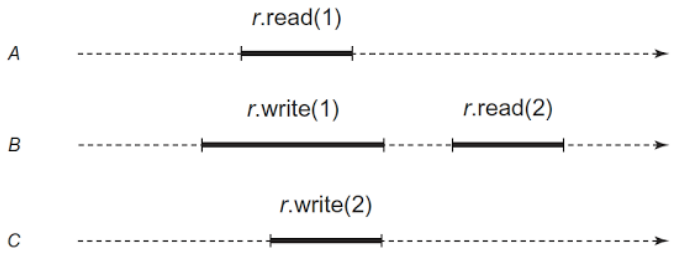



Figure 3.13 First history for Exercise 24.

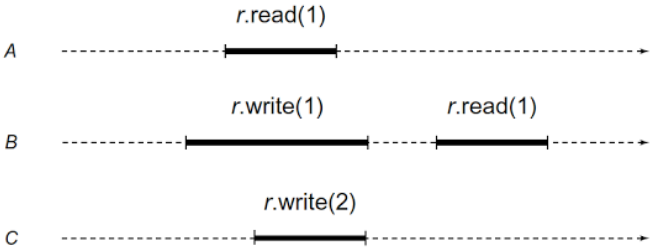



Figure 3.14 Second history for Exercise 24.

**Exercise 25.** If we drop condition L2 from the linearizability definition, is the resulting property the same as sequential consistency? Explain.

Sim, pois a diferença entre linearizability e sequential consistency é de que no primeiro a característica de tempo real é preservada. Removendo L2 as chamadas de método podem se sobrepor, apenas dando o resultado esperado.

**Exercise 27.** The AtomicInteger class (in the java.util.concurrent.atomic package) is a container for an integer value. One of its methods is boolean compareAndSet(int expect, int update).
This method compares the object's current value to expect. If the values are equal, then it atomically replaces the object's value with update and returns true. Otherwise, it leaves the object's value unchanged, and returns false. This class also provides int get() which returns the object's actual value.

Consider the FIFO queue implementation shown in Fig. 3.15. It stores its items in an array items, which, for simplicity, we will assume has unbounded size. It has two AtomicInteger fields: head is the index of the next slot from which to remove an item, and tail is the index of the next slot in which to place an item. Give an example showing that this implementation is not linearizable.

```
1  class IQueue<T> {
2    AtomicInteger head = new AtomicInteger(0);
3    AtomicInteger tail = new AtomicInteger(0);
4    T[] items = (T[]) new Object[Integer.MAX_VALUE];
5    public void enq(T x) {
6      int slot;
7      do {
8        slot = tail.get();
9      } while (! tail.compareAndSet(slot, slot+1));
10     items[slot] = x;
11   }
12   public T deq() throws EmptyException {
13     T value;
14     int slot;
15     do {
16       slot = head.get();
17       value = items[slot];
18       if (value == null)
19         throw new EmptyException();
20     } while (! head.compareAndSet(slot, slot+1));
21     return value;
22   }
23 }
```

Figure 3.15 IQueue implementation.

deq() sempre vai levantar exception pois tail vai começar a preencher a partir da posição 1.

*É possível ter uma thread que completou a execução da linha 10 para a inclusão do valor x e não terminou a execução do método ainda. Nesse momento, é possível que uma outra thread consiga passar pelo deq, obtendo o valor x antes mesmo de o método de inclusão ter sido concluído.*

**Exercise 28.** Consider the class shown in Fig. 3.16. According to what you have been told about the Java memory model, will the reader method ever divide by zero?

```
1  class VolatileExample {
2    int x = 0;
3    volatile boolean v = false;
4    public void writer() {
5      x = 42;
6      v = true;
7    }
8    public void reader() {
9      if (v == true) {
10       int y = 100/x;
11     }
12   }
13 }
```

Figure 3.16 Volatile field example from Exercise 28.

Não. Pode acontecer de o método reader() ser chamado de forma a dividir por zero pelo fato de a variável x não ser volatile. Entretanto, uma vez que a operação do construtor seja executada, x deverá ser tratada como 42.