1. Se da urmatoarea implementare de referinta pentru algoritmul de lista coarse-grained in care protejarea operatiilor asupra listei se realizeaza printr-un lock singular membru al listei utilizat de fiecare metoda: CoarseList.java. Identificati cate o linie de cod din metodele add si remove care corespunde punctelor de linearizare pentru situatiile:
   a) adaugare element cu succes in lista;
   b) esec la adaugare element in lista;
   c) stergere element cu succes din lista;
   d) esec la stergere element din lista.
   Argumentati raspunsul.

   a) `pred.next = node;` deoarece este momentul în care lista se modifică prin adăugarea unui nou nod (pred.next nu va mai fi current).

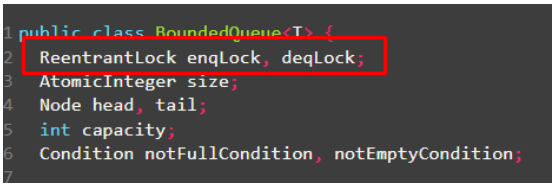
   b) `return false;` când apelul funcției add returnează false (dacă elementul primit ca argument este deja în listă), efectul metodei devine vizibil în exteriorul funcției, mai exact că noul element nu poate fi adăugat.

   c) `pred.next = current.next;` aceeași explicație ca la punctul a, cu precizarea că înaintea apelului pred.next avea valoarea current.

   d) `return false;` la fel ca la punctul b, lista nu se modifică pentru că elementul nu este în listă (key != current.key)

   Punctul de linearizare este instrucțiunea atomică din execuția unei metode a cărui efect este vizibil și în exteriorul metodei. La modul general, pentru fiecare situație se poate considera și `lock.unlock();` punct de linearizare deoarece se modifică starea lacătului, iar deblocarea lui indică ieșirea din secțiunea critică. Cu toate acestea, execuția instrucțiunii nu indică dacă un element a fost adaugat/șters cu succes sau nu.

2. **a) Observam ca membrul size este de tip AtomicInteger, ceea ce presupune ca metodele de incrementare, respectiv decrementare sunt atomice. In aceste conditii, in pseudocodul de mai sus: Mai este necesara in metoda enq plasarea size.getAndIncrement() în cadrul secțiunii protejate de enqLock? Argumentati.**



Se observa faptul ca, deși metoda *getAndIncrement* oferă sincronizare variabilei *size*, aceastea, spre deosebire de *enqLock.lock(),* nu ofera fairness. *enqLock* este de tip ReentrantLock, deci threadurile vor avea acces la variabila în ordinea în care au încercat să o modifice. Acest fairness este necesar pentru rularea corespunzătoare a pseudocodului.

In concluzie, este necesara in metoda *enq* plasarea *size.getAndIncrement()* în cadrul secțiunii protejate de *enqLock.*

b) Presupunem ca în clasa internă Node ar exista un membru ReentrantLock nodelock (similar cu lacatul din structura unui nod individual din lista fine-grained), și ca ne folosim de acest lacat din nodul head în locul enqLock și respectiv de lacatul din nodul tail in locul deqLock, ca mai jos: Va mai functiona în acest caz metoda enq corect păstrând caracterul FIFO al cozii? Argumentati.

Metoda enq păstrează caracterul FIFO al cozii atunci când avem mai multe thread-uri care o apelează simultan deoarece lacătul aplicat nodului head nu permite introducerea altor noduri înainte de a finaliza adăguarea nodului curent (se adaugă elementele în ordinea apelării funcției), iar nodul head nu va fi modificat niciodată în funcția enq.

c) Presupunem ca in loc de utilizarea conditiilor notFullCondition si notEmptyCondition, si a flagurilor mustWakeDequers, respectiv mustWakeEnqueuers, pentru notificari intre threaduri, metodele enq si deq se vor folosi pur si simplu de o operatie de spinning, ca mai jos:

```java
public void enq(T x) {
        boolean mustWakeDequeuers = false;

        enqLock.lock();
        try {
                while (size.get() == capacity) {}; //spinning
                Node e = new Node(x);
                tail.next = e;
                tail = tail.next;
                size.getAndIncrement();
        } finally {
                enqLock.unlock();
        }
  }

  public T deq() {
        boolean mustWakeEnqueuers = false;
        T v;

        deqLock.lock();
        try {
                while (head.next == null) {}; //spinning
                v = head.next.value;
                head = head.next;
                size.getAndDecrement();
        } finally {
                return v;
                deqLock.unlock();
        }

  }
```

Va mai functiona în acest caz algoritmul pentru coada corect (ignorând scăderile în performanță)?

Raspuns:

În cazul de față, algoritmul funcționează corect cu spinning deoarece thread-ul se blochează la instrucțiunile while din ambele metode si înaintează cand va fi eliminat un element din coada în cazul lui enq() sau adauga un element in coadă, iar thread-urile se așteaptă unul pe celălalt din cauza lock-ului care se pune la ambele metode.

Ce s-ar intampla dacă s-ar amesteca cele două abordări în modul următor?:

```
public void enq(T x) {
        boolean mustWakeDequeuers = false;

        enqLock.lock();
        try {
                while (size.get() == capacity) {
                        notFullCondition.await();
                }
                Node e = new Node(x);
                tail.next = e;
                tail = tail.next;
                if (size.getAndIncrement() == 0) {
                        mustWakeDequeuers = true;
                }
        } finally {
                enqLock.unlock();
        }

        if (mustWakeDequeuers) {
                deqLock.lock();
                try {
                        notEmptyCondition.signalAll();
                } finally {
                        deqLock.unlock();
                }
        }
 }

 public T deq() {
        boolean mustWakeEnqueuers = false;
        T v;

        deqLock.lock();
        try {
                if (head.next == null) {
                        notEmptyCondition.await();
                }

                while (size.get() == 0) {}; //spinning

                v = head.next.value;
                head = head.next;
                if (size.getAndDecrement() == capacity) {
                        mustWakeEnqueuers = true;
                }
        } finally {
                deqLock.unlock();
        }

        if (mustWakeEnqueuers) {
                enqLock.lock();
                try {
                        notFullCondition.signalAll();
                } finally {
                        enqLock.unlock();
                }
        }

        return v;

 }
```

Raspuns:

Cand metoda enq va fi apelata algoritmul va rula normal dar în momentul în care se apelează metoda deq instrucțiunea while va fi ignorată deoarece thread-ul așteaptă să se adauge un element în coadă la instrucțiunea "notEmptyCondition.await();" astfel la momentul cand se ajunge la while, size va fi mereu mai mare decât 0 deoarece decrementarea size-ului se face mereu sub while, și astfel nu se va intra niciodată în bucla while.

d) Se da pseudocodul de mai jos pentru o coada lock-based de aceasta data in varianta nelimitata. Este necesar ca verificarea pentru coada nevida din metoda *deq()* sa fie neapărat plasată în secțiunea protejata prin lock sau ar putea fi plasată și în afara secțiunii protejate prin lock? Argumentati.

```java
public class UnboundedQueue<T> {
  ReentrantLock enqLock, deqLock;
  Node head, tail;


  public void enq (T value) {
        enqLock.lock();
        try {
                Node newNode = new Node(value);
                tail.next = newNode;
                tail = newNode;
        } finally {
                enqLock().unlock();
        }
  }

  public T deq() throws Exception {
        T result;
        deqLock.lock();
        try {
                if (head.next == null) {
                        System.out.println("queue empty");
                        throw new Exception();
                }
                result = head.next.value;
                head = head.next;
        } finally {
                deqLock().unlock();
        }
        return result;
  }

  public UnboundedQueue() {
    head = new Node(null);
    tail = head;
    enqLock = new ReentrantLock();
    deqLock = new ReentrantLock();
  }

  protected class Node {
        public T value;
        public Node next;
        public Node (T value) {
                this.value = value;
                next = null;
  }
}
```

Raspuns:

Verificarea pentru coada nevida din metoda *deq()* nu poate fi și în afara secțiunii protejate prin lock deoarece numărul thread-urilor care vor trece de verificare (în cazul în care am plasa verificarea înafara secțiunii protejate prin lock) va fi mai mare decat numărul de elemente care se afla în coadă și astfel se va produce o eroare.