1.

a. pred.next = node ;

Pe ramura else, se va adauga elementul, iar la linia respectiva va fi punctul de liniarizare pentru ca *pred* este o instanta existenta a structurii si va primi ca si *next* noul nod creat. La randul lui va avea nodul *current* ca si nod succesor.

b. return false;

Noul item nu este adaugat in lista in momentul cand el este deja existent. Deci linia de mai sus reprezinta punctul de liniarizare.

c. pred.next = current.next

Un element este sters din lista daca el este existent, in acel caz, predecesorul lui va lua ca succesor valoarea succesorului nodului curent(cel care vrem sa-l stergem).

d. return false;

Un element nu este sters atunci cand el nu exista in lista, asa ca executia va intra pe ramura else.

2.

a. Este necesara plasarea metodei *getAndDecrement* deoarece, daca lista este plina, noi notificam threadurile ca poata sa faca enqueue, insa daca nu decrementam *size* niciunul nu va putea face enqueue intrucat se va bloca executia in bucla *while* din *enq* intrucat *size == capacity*.

b. Stim ca atunci cand un thread face lock pe o instanta a unui nod, un alt thread oarecare, nu va putea face modificari succesorului instantei respective. Asa incat, de exemplu, daca doua threaduri vor face simultan dequeue, atunci executia lor va putea trece de linia unde s-a facut lockul(*tail.nodelock.lock();*), ele vor putea avea aceeasi valoare *v*, iar head ar putea lua valoarea head.next.next la final. Deci dequeue nu va functiona corect.

c.1

Algoritmul va functiona corect deoarece fiecare thread are acces mutual exclusiv la zona critica. In cazul in care lista este goala/plina iar un thread face deq/enq, threadul va astepta intr-o bucla while pana cand se va adauga/sterge un element.

c.2

Putem observa ca singura diferenta este inlocuirea buclei *while* cu *if* si adaugarea dupa acesta operatia de spinning. Un thread oarecare ce este notificat si care obtine lockul, chiar daca lista este goala (presupunand ca acel element a fost consumat de alt thread) va astepta in bucla while pana cand un element va fi adaugat in lista nepermitand altor threaduri sa obtina lockul.

d.

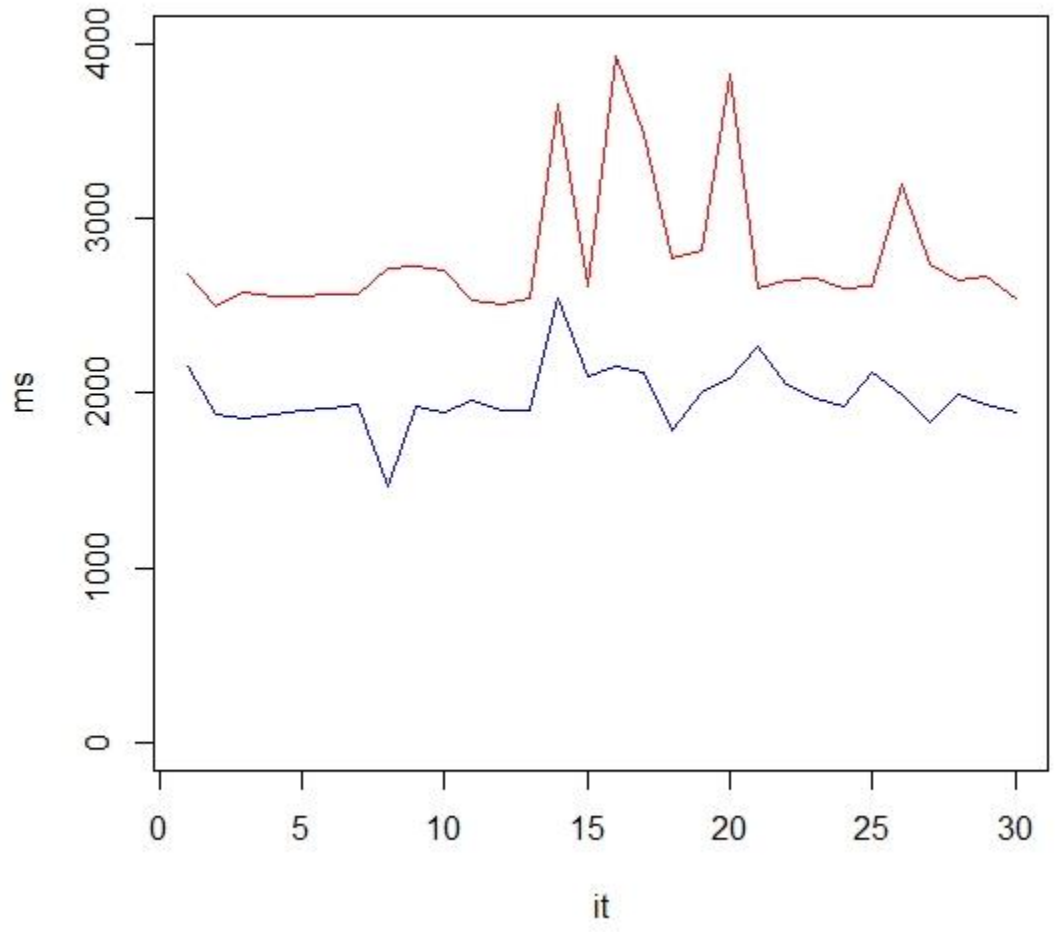
Este necesar ca verificarea sa se realizeze in interiorul lockului deoarece daca ar fi in afara putem obtine un astfel de comportament: daca avem un element in coada si doua threaduri care fac dequeue, ambele threaduri pot trece de verificarea pentru coada nevida, rezultatul fiind ca numai unul din ele va face dequeue iar celalalt va incerca sa acceseze coada vida, spre deosebire de varianta cu if in interior in care, dupa ce primul thread face dequeue, al doilea va arunca o exceptie inainte de accesarea listei goale.

3.

Pentru eficientizarea operatiei de validare a listei ne putem folosi de introducerea unei versionari astfel: in momentul cand lista se modifica, vom incrementa un numar care va reprezenta „versiunea” listei. Acesta numar poate fi un AtomicInteger. In loc sa verificam validarea la fiecare operatie de modificare, putem pune o conditie precedenta acesteia care sa verifice daca versiunea de la inceputul apelului corespunde cu versiunea inainde de modificarea efectiva a listei. In cazul in care aceste doua versiuni sunt egale, asta inseamna ca lista nu a fost modificata si se poate evita parcurgea din nou a listei, iar daca nu, ne vom folosi de validare.

Mai jos sunt doua grafice care reflecta timpul de executie pentru: lista modificata (albastru) si lista originala (rosu). Pentru a obtine o medie relavant statistica, a fost repetat procesul de 30 de ori.

Pentru comparare, s-au adaugat 100000 de elemente folosind 4 threaduri.



Media duratei de executie a listei modificate: 1977ms

Media duratei de executie a listei originale: 2793ms