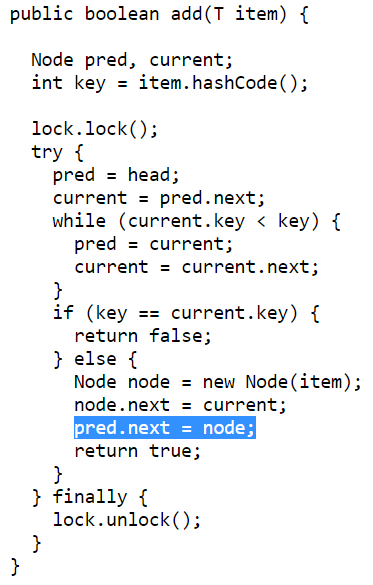
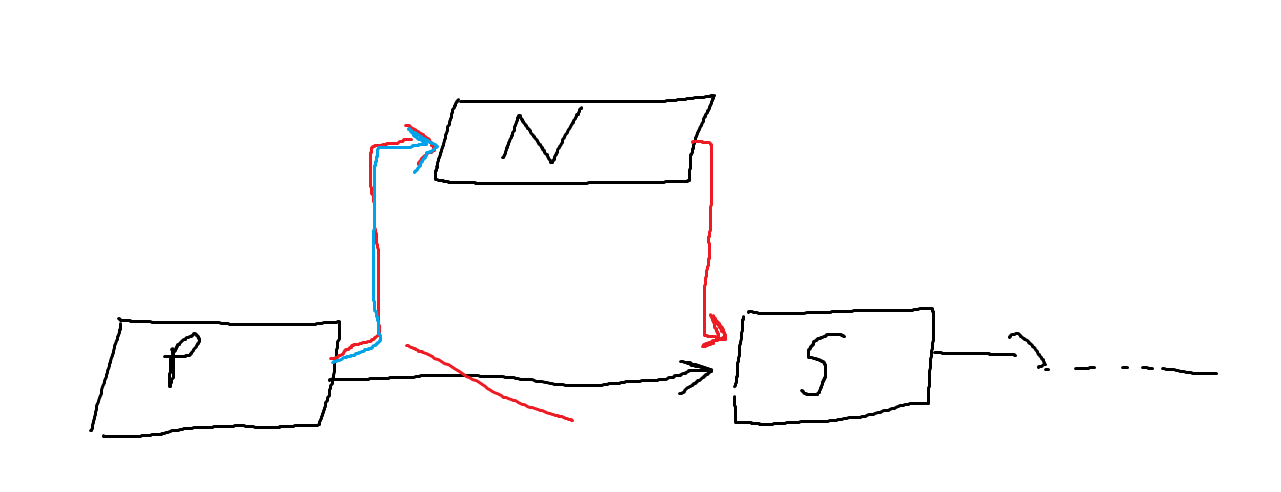
Exercitiul 1

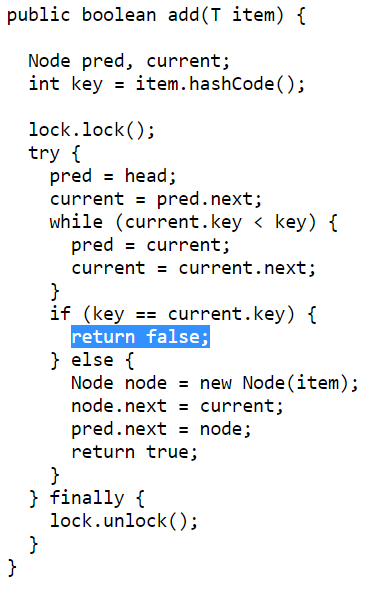
1. Adaugare cu succes in coada:



Argument: Se creeaza o noua legatura de la noul nod creat la succesorul sau. Legatura de la predecesor se actualizeaza la noul nod creat. In momentul actualizarii legaturii predecesorului, in lista s-a schimbat componenta.

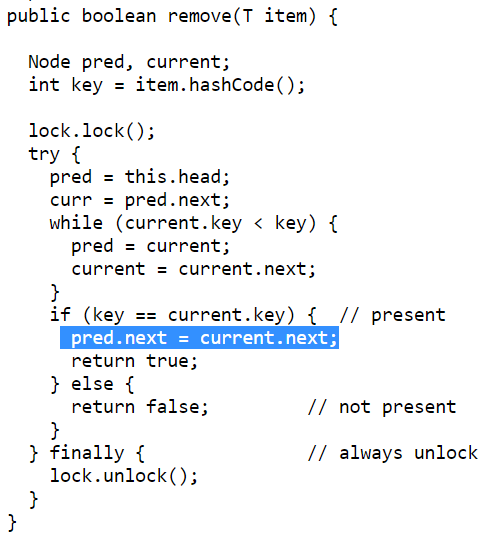


1. Adaugare cu esec in coada

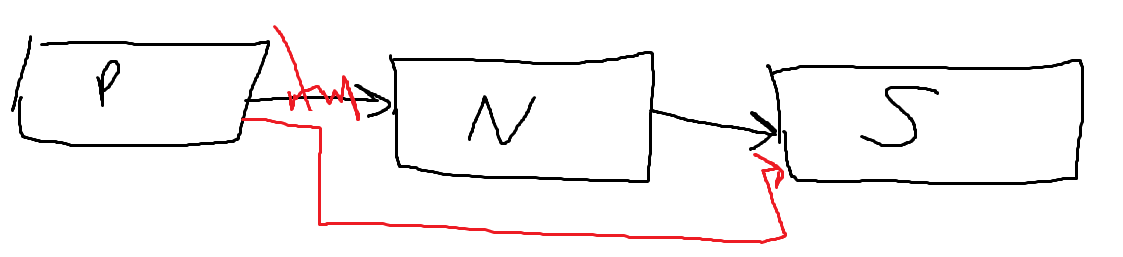


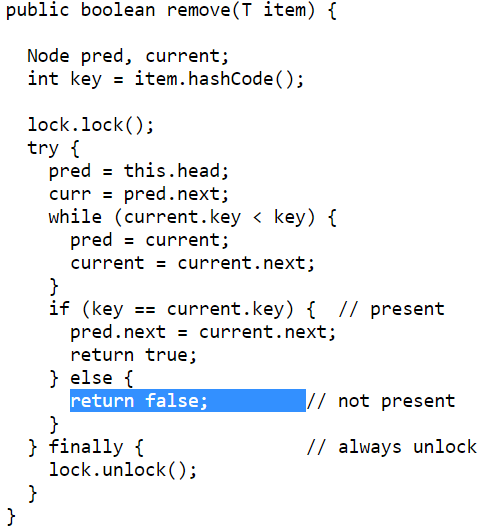
Argument: Se observa ca elementul respectiv care se doreste a fi adaugat in coada exista deja, prin urmare nu mai are rost inserarea acestuia inca o data. Acest lucru se semnealeaza cu returnarea valorii false. Conditia din if este verificata in orice scenariu (cand elementul este sau nu in coada), prin urmare testarea ei nu reprezinta punct de linearizare.

1. Stergere cu succes din coada



Argument: In cazul gasirii elementului ce este dorit a fi sters, se schimba legatura dintre predecesorul sau cu succesoru. In momentul schimbarii legaturii, se schimba si componenta listei, iar vechiul nod nu mai este in lista/referit de un alt nod din lista.



1. Stergere cu esec din coada

Argument: Dupa ce am parcurs intreaga lista, in cazul in care valoarea variabilei key nu corespunde cu valoarea obiectului cautat pentru stergere, se semnaleaza faptul ca nu se poate sterge obiectul deoarece nu exista acesta in lista prin returnarea valorii false.

Exercitiul 2

1. Nu, nu este necesar apelul la size.getAndIncrement in zona protejata de lacat. Motivul este ca un thread la un moment dat va executa operatiunea de size.getAndIncrement(), fiind o operatie atomica. Putem pune apelul metodei dupa ce se elibereaza lock-ul astfel:

finally {

head.nodelock.unlock();

}

if (size.getAndIncrement()==0) {

tail.nodelock.lock();

try {

notEmptyCondition.signalAll();

} finally {

tail.nodelock.unlock();

}

}

Metoda getAndIncrement va fi apelata de orice thread care a adaugat (enquer thread). Dintre 2 thread-uri care adauga, unul va apela primul acesta si va vedea valoarea 0, trezind thread-urile deque, daca sunt, care asteapta inserarea unui element pentru scenariul in care lista este goala.

1. Operatiune de enque va functiona pastrand caracterul FIFO.

Argumentare: Toate thread-urile care vor sa faca enque pe lista se vor comporta ca si cum ar trebui sa obtina lock-ul enqlock din versiunea anterioara, deoarece fiecare thread enquerer trebuie sa obtina lock-ul pe head.

Acestea lucreaza pe capatul cozii, unde insereaza fiecare, pe rand, cate un element. Din aceasta cauza nu se interpun elementele ce trebuie adaugate de catre un thread enquerer cu elementele deja existente in lista.

Capatul cozii (tail) este procesat de catre un singur thread (enquer) la un moment dat. Asemanator si pentru head cu dequer.

Pentru o lista cu mai mult de 0 elemente, adaugarea si inserarea se executa fara sa se influenteze, la capete diferite.

Pentru o coada goala, se executa intai inserarea si apoi stergerea.

Daca coada este plina, un thread enquer va lua lock pe head si va astepta eliberarea cozii, iar celalalte enquerer vor tot incerca sa obtina lock-ul. Dupa notificarea de catre un thread dequer, enquer-ul ce are lock-ul va insera, iar urmatorul enquer va vedea dupa ce obtine lock-ul faptul ca lista este plina si va astepta.

1. Pentru varianta cu spinning:

Da, algoritmul va continua sa functioneze corect.

Toate operatiunile de enque si deque se executa o data , un thread executand un singur enque datorita lock-lui pentru enque si un thread executa un singur deque tot datorita lock-lui pentru deque (nu vom avea mai multe operatiune de enque sau deque de o data).

Spinning-ul va fi realizat de catre un singur thread pentru enque si deque, celalalte thread-uri, daca sunt, vor astepta sa obtina lock-ul pentru respectiva operatiune.

Pentru o lista cu mai multe noduri, deoarece se opereaza la capete diferite, nu se afecteaza operatiunile intre ele. Daca lista este plina, se executa doar o operatiune de deque care actualizeaza head-ul. Thread-ul pentru enque face in acest timp spinning.

Daca lista este goala, un thread care vrea sa faca deque preia lock-ul, celalalte asteapta eliberarea lock-lui. Cel cu lock va astepta ca nodul head sa aiba o valoarea urmatoarea diferita de null.

Pentru varianta combinata:

Thread-urile care vor face enque se vor comporta la fel ca in varianta originala a programului.

Diferenta substantiala apare in metoda deque, unde avem verificarea conditiei ca head.next sa fie diferit de null, in loc de while(head.next==null).

Bucla while care era initial este practic sparta in 2 secvente de cod:

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

si

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

Un thread care face deque poate vedea ca lista este goala si face await pe conditie. Pana se adauga mai multe elemente, pot aparea mai multe thread-uri de deque care sa faca deque si care vor ajunge sa faca await. La inserarea unui element, un thread deque va iesi din await, va vedea ca size-ul cozii este mai mare decat 1 si va scoate un element. Dupa, un al doilea thread deque va iesi din await si trebuie sa verifice daca mai este vreun element in coada. Cum nu mai este altul, va face spinning. Daca nu are aceasta instructiune while pusa dupa conditie, al doilea thread deque ar fi incerca sa scoata un element null din coada.

1. Singurul loc unde ar fi logic sa facem verificarea cozii ar fi inaintea secventei protejate de lock.

Problema este ca nu putem sa scoatem verificarea din zona protejata de lock.

Fie scenariul in care coada are un element si vin 2 thread-uri care vor sa faca deque.

Ambele thread-uri vor vedea ca mai este un element, dar doar unul va scoate elementul din coada, celalalt thread va incerca sa scoata un element null, deoarece lista va fi goala. Aceasta actiunea va cauza aruncarea unei exceptii.