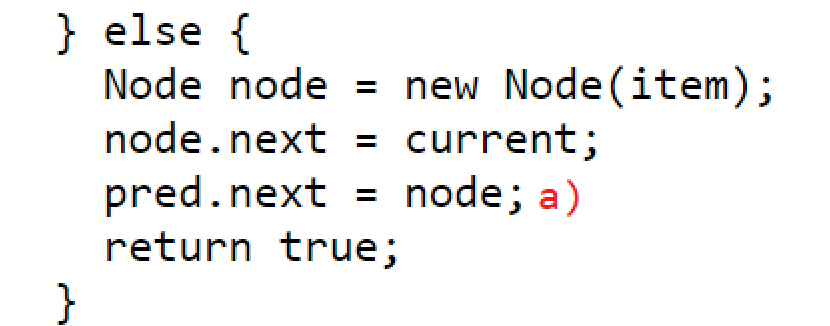
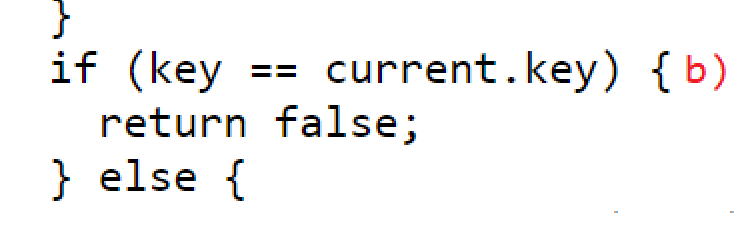
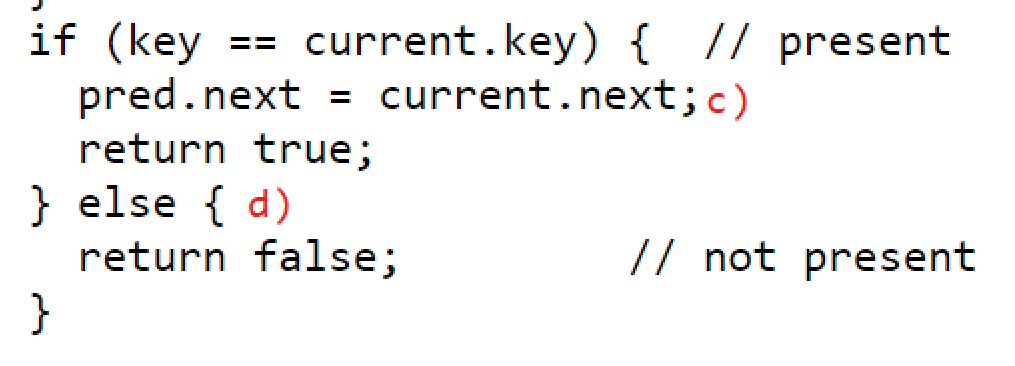
Tema 2 TPM

Echipă: Inculeț Alexandru, Inculeț Bogdan, Ioan Alexandra

Ex1: punctele de linearizare sunt următoarele:







Acestea sunt puncte de linearizare deoarece:

1. Adăugarea este realizată cu succes chiar după ce elementul *pred.next* va lua valoarea lui *node.*
2. Adăugarea elementului în coadă eșuează în if-ul *key == current.key.*
3. Eliminarea unui element din coadă are succes dacă *key == current.key* și are efect chiar după linia *pred.next = current.next.*
4. Altfel, eliminarea elementului eșuează iar punctul de linearizare este chiar în *else*.

Ex2.

1. Da, este necesară plasarea size.*getAndIncrement()* în cadrul secțiunii protejate de enqLock. Din cauza faptului că există o capacitate maximă a cozii, dacă ar fi să incrementăm variabila atomică după ce facem unlock la enq, atunci ar fi posibil ca alt thread care a făcut enq să ia lock-ul și să adauge și el încă o variabilă, crezând că dimensiunea cozii este încă 0. Astfel s-ar introduce mai multe elemente în coadă decât capacitatea ei.





1. Da, metoda *enq* va funcționa corect deoarece lock-ul din head va face indisponibilă operația *deq* și *enq* pentru alte threaduri în timp ce threadul curent introduce un element nou în coadă, analog și pentru lock-ul din tail.
2. Da, algoritmul pentru coadă va mai funcționa corect și cu metoda spinning, deoarece, niciun thread nu poate să facă *enq* în același timp cu alte threaduri, și de asemenea niciun thread nu poate să facă *deq* în același timp cu alte threaduri care fac aceiași operație.

Pe lângă asta, dacă ar exista 2 threaduri, unul care face *enq* și altul care face *deq* în același timp, cel cu *deq* ar aștepta pentru cel cu *enq* să introducă un element, mai exact ar aștepta până când se execută linia de cod : *tail.next = e;* . Dacă ipotetic thread-ul *deq* ar lua-o înfață, odată ce a fost eliberat de *spinning,* și ar apuca să deblocheze *lock*-ul inainte de thread-ul *enq,* acesta ar fi mutat *head*-ul în față cu o poziție, ajungând ca *head.next* să fie iar *null*, nelăsând alte eventuale *thread-*uri *deq* să mai scoată vreun element. Și de asemenea, deoarece *size* este o variabilă atomică la finalul executiei simultane a 2 threaduri ( unul cu *enq* si celalalt cu *deq*), aceasta rămăne actualizată corect.

Algoritmul funcționează și cu cele 2 metode amestecate, deoarece deși în apelul *deq* este înlocuit *while*(*head*.*next*==*null*…. cu *if* condiția este verificată adițional dedesubt cu ajutorul variabilei atomice *size,* prin apelul *size.get()*, rezultatul fiind unul identic cu metoda de deasupra.

1. Da, este necesar ca verificarea pentru coada nevidă să fie în zona critică deoarece daca ar exista mai multe thread-uri *deq* care spre exemplu ar rula simultan asupra unei cozi de dimensiune 1, se poate ca mai multe threaduri să treacă simultan cu succes de verificarea de dinainte de *lock: if(head.next != null)* și să rămână blocate de lock, iar dupa ce elementul va fi scos de coadă de unul dintre thread-uri, deblocând lock-ul, o să intre alt thread în zona critică știind lungimea cozii ca fiind nevidă, rezultând ștergeri de elemente ce nu există și chiar *NullPointerException.*