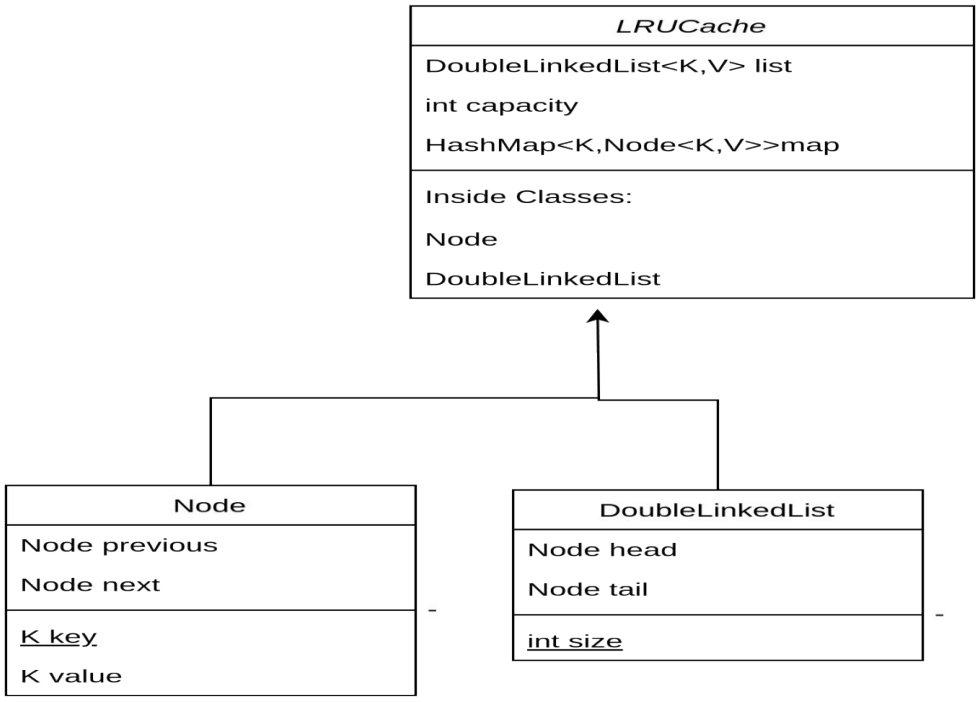
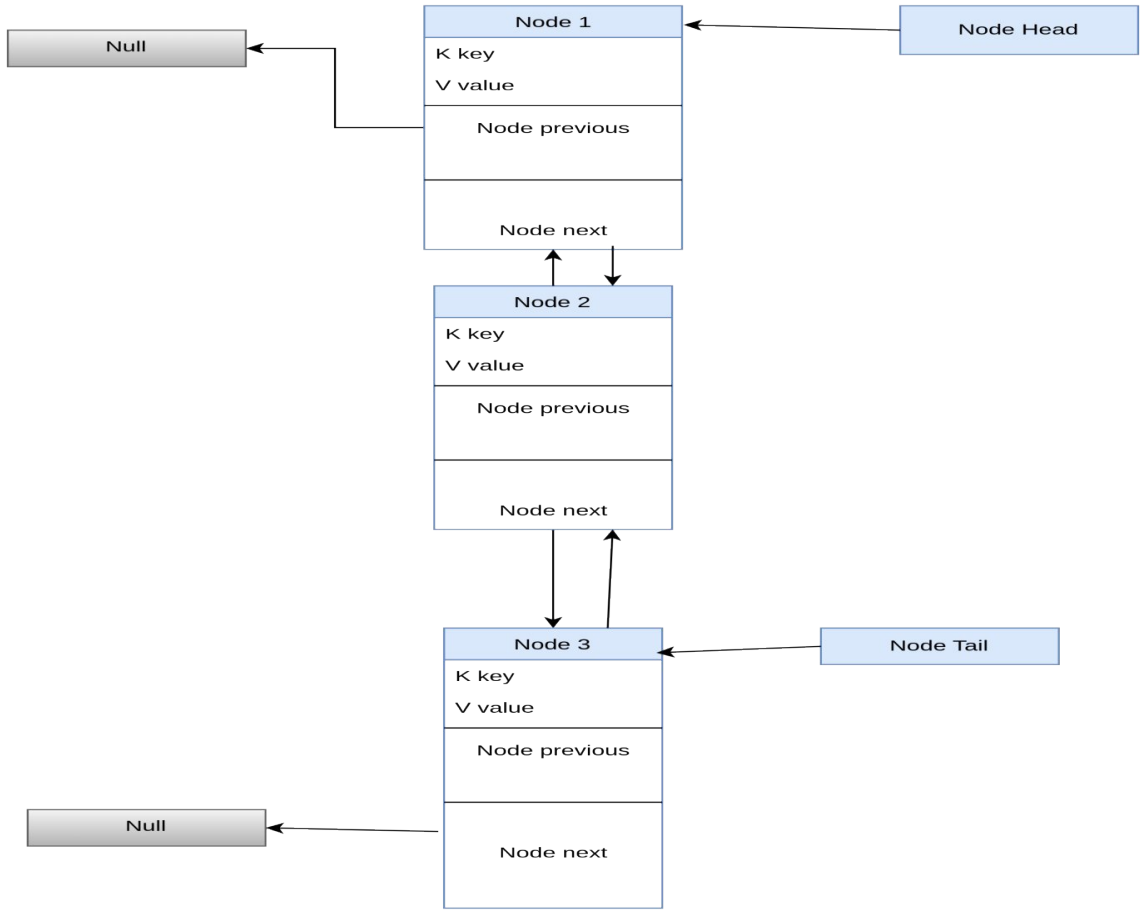
Εσωτερική δομή κλάσης LRUCache:



Παράδειγμα με 3 κόμβους το DoubleLinkedList



Όπου κάθε κόμβος ξέρει τον προηγούμενο κόμβο μέσω του στιγμιότυπου Node previous, και τον επόμενο κόμβο μέσω του Node next. Καθώς και κρατάμε σαν πεδιο της κλασης τον κόμβο που ειναι στην αρχή (head) και τον κόμβο στο τελος(tail), καθως και αυτα ενημερωνονται οταν χρειάζεται στις διάφορες μεθόδους που χρησιμοποιούμε. Επισης σαν πεδιο εχουμε και ενα μεγεθος (size) που αυξάνεται όταν προσθέτουμε κόμβους και αφαιρείται όταν βγάζουμε κόμβους.
Ξεκινάμε με την παραδοχή οτι κάθε λίστα στην αρχή ειναι κενή, με μηδενικό μεγεθος και πάντα το previous του head και το next του tail θα ειναι null.

Το DoubleLinkedList ειναι υπευθυνο για την διαχειριση των κομβων (να προσθεσει/αφαιρεσει κτλπ), ενω το HashMap<K,Node<K,V>> map αποθηκευει καθε κόμβο σύμφωνα με το κλειδί του κόμβου, και έτσι μέσω του κατακερματισμού κάνουμε εύκολη και γρήγορη την αναζήτηση.

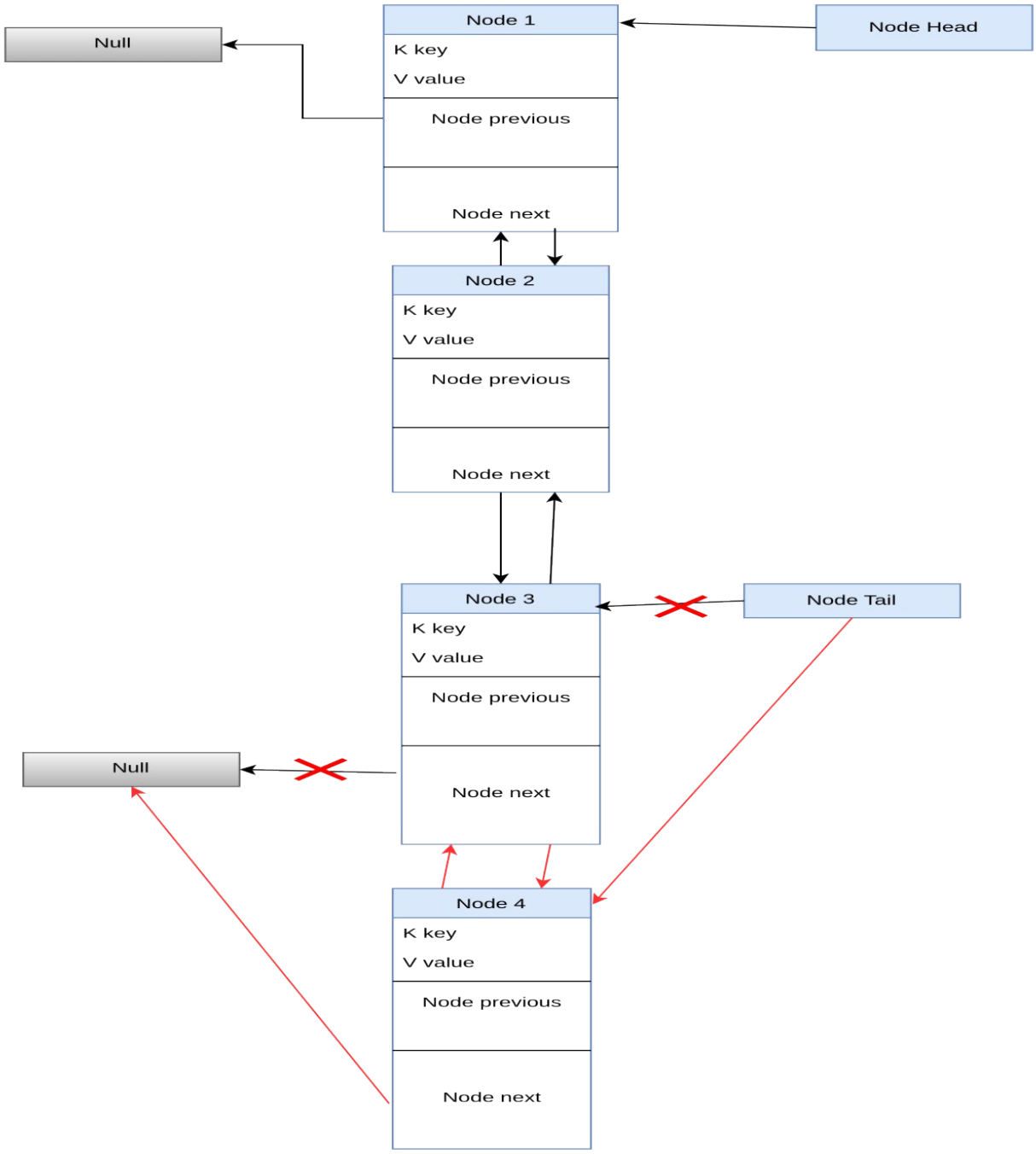
Find	Insert(Last)	Delete(First-least frequent)
Γρήγορο λόγω του κατακερματισμού στο map	Γρηγορο-απλη εισαγωγή στο τέλος και αλλαγή των δεικτών του προηγούμενου κόμβου (λόγος που χρειάζομαι διπλά συνδεδεμένη λίστα)	Γρηγορο- “αποσυνδεση” δεικτή head και δείκτη previous του επομενου κομβου

Ο χρήστης μου προκαθορίζει το μέγιστο μέγεθος κομβων που μπορεί να χωρέσει το LRUCache. Όταν αυτό γεμίσει (δηλαδή το capacity της LRUCache φτάσει το size της λιστας DoubleLinkedList-το οποίο αυξάνεται με την προσθήκη κάθε νέου κόμβου) τότε αφαιρείται το πιο παλιό στοιχείο που είχαμε πρόσβαση, με την μέθοδο deleteFirst.

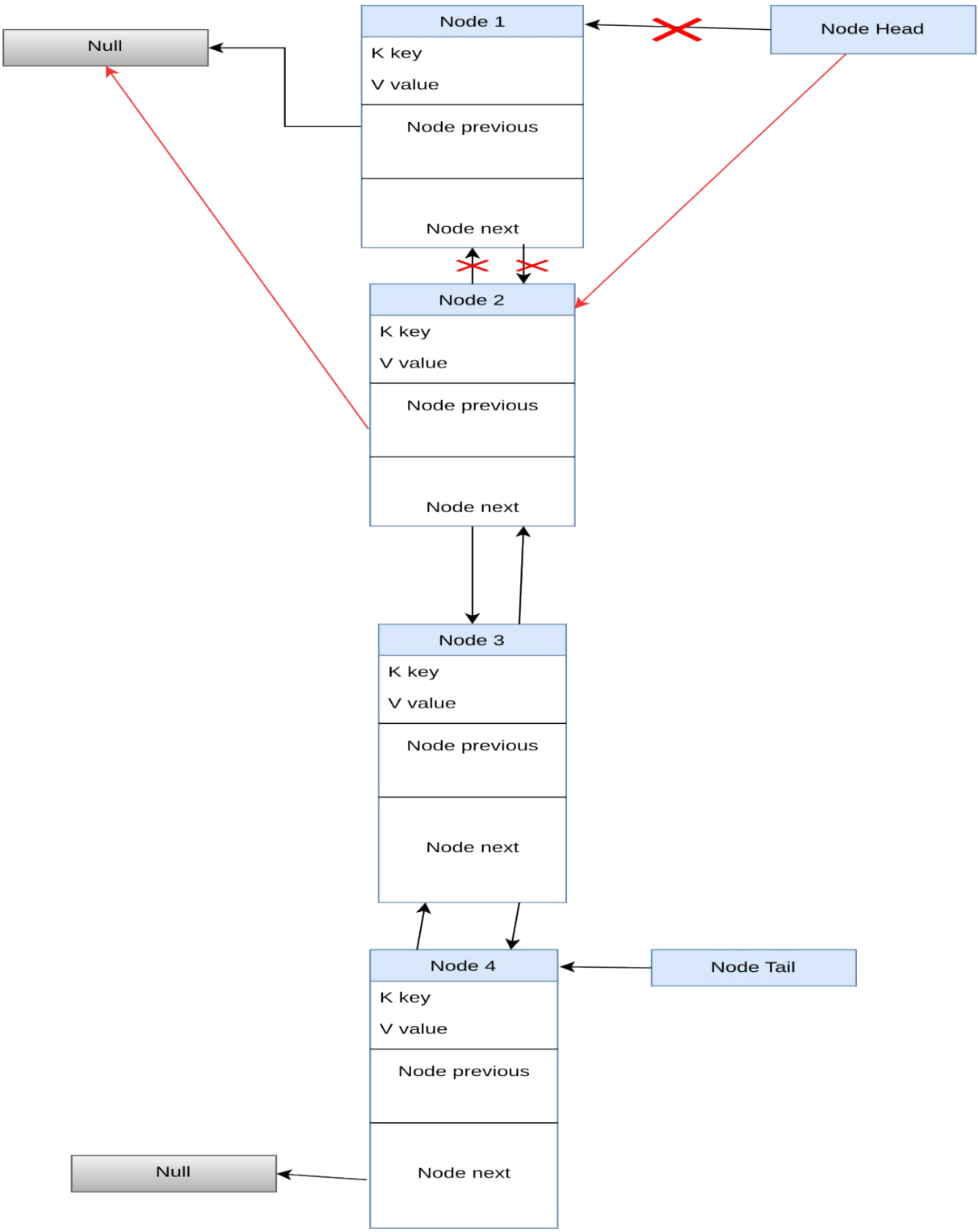
Η μέθεδος get της LRUCache, μας επιστρέφει την τιμή του κόμβου που αντιστοιχεί του κλειδιού που δώσαμε, και τον βάζει το τέλος (tail) της λίστας γιατι χρησιμοποιήθηκε πιο πρόσφατα.

Η μέθεδος put της LRUCache παίρνει ενα συνδυασμό τιμής-κλειδιού και το αποθηκεύει σαν κόμβο στη λίστα. Ελέγχω πρώτα αν έχω ήδη κάποιον αποθηκευμένο κόμβο με αυτό το κλειδί- αν έχω τοτε ανανεώνω την τιμή του και τον βάζω στο τέλος. Αν δεν έχω κόμβο με αυτό το κλειδί, τότε φτιάχνω νέο κόμβο και το προσθέτω στο τέλος της λίστας.

Παράδειγμα προσθήκης κόμβου νο. 4



Παράδειγμα διαγραφής κόμβου



ΜΕΡΟΣ ΔΕΥΤΕΡΟ

Στην μεθοδο remove (Node node): μειωνουμε το μεγεθος της λιστας.
Προσθηκη ελεγχου στις μεθοδους put/addNode(private μεθοδος) ωστε να μην προστιθεται null κομβος. Και στην μεθοδο put ελεγχω αν προσπαθησει να ενημερωσει εναν υπαρχων κομβο που εχει null key με νεο null value.

Το totalOperations αυξανεται μονο στην μεθοδο get (βαση σχεδιαγραμματος εκδωνησης)

```
Total operations: 100000
Cache Hits: 52300
Cache Misses: 47700
Hit Rate: 52.30%
Miss Rate: 47.70%
```

Τα ποσοστα των αποτελεσματων ειναι αναλογα οχι μονο βαση της επιλογης mru/lru αλλα και της αναλογιας capacity με ευρος κλειδιων. Οσο η αναλογια capacity/keyRange μεγαλωνει τοσο αυξανωνται και τα ποσοστα hitRate και στις δυο τεχνικες cache, ενω οσο μειωνεται τοσο πεφτουν. Παρατηρειται ομως σε ολες τις περιπτωσεις υψηλοτερο hitRate για την LRU cache.

Σχεδιασμος main:
Δυο τεχνικες απεικονισης cache, για να συγκριθουν τα αποτελεσματα.
Εχω δυο keyrange: 0-49, το οποιο μεσα σε ολες τις επαναληψεις που θα γινουν εχει 80% να πετυχω κλειδι σε αυτο το range
50-149: εχει 20% πιθανοτητα σε καθε επαναληψη για να τυχει κλειδι εδω.
Αφου βρεθει ψευδοτυχαια το κλειδι, δινω 50% πιθανοτητα με το random.nextBoolean() για να κανω put ή get.
Αφου φτασουμε τα 100.000 get (οπως ζηταει η εκφωνηση), εμφανιζουμε τα στατιστικα στοιχεια για καθε τεχνικη.

Σχεδιασμος test:
Ιδιες μεθοδοι με πρωτο μερος, ελεγχουν και τις δυο πολιτικες cache.

ΜΕΡΟΣ ΤΡΙΤΟ

- Προσθηκη ακεραιας μεταβλητης (int timesUsed) σε εσωτερικη κλαση Node
- Προσθηκη δομης δεδομενων TreeMap με key Node<K,V> και με value Integer (οπου θα ειναι ουσιαστικα το timesUsed του κλειδιου), καθως και εσωτερικη κλαση Comparator ωστε να ταξινομει την TreeMap βαση του πεδιου timesUsed των Nodes (με αυξουσα σειρα).

-Αν εισαχθουν δυο entries στο treeMap που εχουν το ιδιο value(Integer,δηλαδη συχνοτητα), οπως και θα συμβαινει στην αρχη, αντι να γυριζει ο Comparator 0 (λαθος γιατι αν γυριζει 0 θα θεωρει τα δυο entries ιδια, αλλα η δομη Map δεν υποστηριζει διπλοτυπα) γυριζει την διαφορα των hashcode των κλειδιων των Node.

π.χ.
int capacity=4;
MyCache <Integer,String> exampleCache= new MyCache<>(capacity, CacheReplacementPolicy.LFU);

exampleCache.put (1 ,"ExampleNode");
//TreeMap={ Node(1,"ExampleNode")=1 }

exampleCache.put (2 ,"Example2");
//TreeMap={ Node(1,"ExampleNode")=1, Node(2,"Example2")=1 }

//(εδω γινεται ετσι η τοποθετηση λογω των hashcode των κλειδιων που εχουν οι δυο κομβοι)
exampleCache.get(1);
//TreeMap= { Node(2,"Example2")=1, Node(1,"ExampleNode")=2 }

exampleCache.put (0 ,"0node");
//TreeMap= { Node(0, "0node"), Node(2,"Example2")=1, Node(1,"ExampleNode")=2 }

exampleCache.put (5 ,"5node");
//TreeMap= { Node(0, "0node")=1, Node(2,"Example2")=1, Node(5 ,"5node")=1, Node(1,"ExampleNode")=2 }
//Εδω το node5 μπαινει μετα το Node2, διοτι εχουν την ιδια συχνοτητα (κριτηριο Comparator), αλλα το hashCode του node5 ειναι μεγαλυτερο.

exampleCache.put (2 ,"Example2.1");
//TreeMap= { Node(0,"0node")=1,Node(5 ,"5node")=1, Node(1,"ExampleNode")=2, Node(2 ,"Example2.1")=2}

-Για την διατηρηση των στοιχειων εντος του TreeMap: Οταν θελω να ανανεωσω καποιο entry, διαγραφω το παλιο entry κρατωντας τα στοιχεια του, και το ξαναεισαγω με τα νεα του δεδομενα, καθως και αυξανω την timesUsed κατα ενα (μεθοδος incrementFreq).
Ενώ παραλληλα συνεχιζω να ενημερωνω κανονικα τις αλλες δυο δομες δεδομενων μου (HashMap και DoubleLinkedList)

```
}else{
    //εαν υπαρχει κομβος με αυτο το key
    //ενημερωση του χαρτη
    existNode.value=(V)value;
    list.moveToLast(existNode);
    if (policy==CacheReplacementPolicy.LFU){
        treeMap.remove(existNode);
        existNode.incrementFreq();
        treeMap.put(existNode, existNode.getTimesUsed());
    }


}
```

Επειδη το TreeMap χρειαζεται μονο στην περιπτωση που εχω LFU policy, το χρησιμοποιω μονο μεσα σε if statement ωστε να μην σπαταλουνται ποροι στην περιπτωση που εχω καποιο αλλο policy.

Σχεδιασμος Main:
Ιδιος με πριν, απλη προσθηκη ενος ακομα policy και cache για την απεικονιση αποτελσματων.

Σχεδιασμος test:
Ελεγχοι και για τις τρεις πολιτικες τις ιδιες μεθοδους του πρωτου μερους.