Αναφορά πρώτου παραδοτέου Δομών Δεδομένων. **Καθηγητής: Δημήτριος Μιχαήλ**

Υλοποίηση Cache με στρατηγική LRU(least recently used), χρησιμοποιώντας διπλά συνδεδεμένη λίστα και πίνακα κατακερματισμού.

Επαρκείς έλεγχοι μέσω JUnit tests για την σωστή λειτουργία και την συνέπεια στη στρατηγική της μνήμης.

Ομάδα 33.

Μέλη:

it2021155 Αιμίλιος Παπακωνσταντίνου **it21535 Νικόλαος Νικηφόρος**

Βασική Στρατηγική:

Για την υλοποίηση φτιάξαμε μια δική μας διπλά συνδεδεμένη λίστα CustomLinkedList με dummy κόμβους head-tail και τους συνδέσαμε. Ορίσαμε την κλάση Node για την αναπαράσταση κάθε κόμβου της λίστας.

Συνδυάσαμε την LinkedList που φτιάξαμε με πίνακα κατακερματισμού για να έχουμε χρόνο προσπέλασης O(1).

Στην LRU η προσπέλαση ή καταχώρηση ενός στοιχείου το τοποθετεί στην αρχή της λίστας. Εάν η μνήμη γεμίσει τότε το τελευταίο στοιχείο, άρα το στοιχείο που χρησιμοποιήσαμε παλιότερα, αφαιρείται.

Αναπτύξαμε δοκιμές junit tests για τον έλεγχο της υλοποίησης μας σε idle και edge καταστάσεις.

Υλοποίηση κόμβου Node:

Η κλάση Node είνα η βάση για την διπλά συνδεδεμένη λίστα μας.

Περιέχει:

1. Το κλειδί Key.
2. Την τιμή Value που αποθηκεύεται.
3. Δείκτες next και prev για τον επόμενο και προηγούμενο κόμβο.

Node.java

```java
package org.hua.cache;

public class Node<K, V> {
    K key;           // Το κλειδί του κόμβου
    V value;         // Η τιμή του κόμβου
    Node<K, V> prev; // Δείκτης στον προηγούμενο κόμβο
    Node<K, V> next; // Δείκτης στον επόμενο κόμβο

    public Node(K key, V value) {
        this.key = key;
        this.value = value;
    }
}
```

Υλοποίηση διπλά συνδεδεμένης λίστας:

Για την CustomLinkedList χρησιμοποιούμε δύο dummy κόμβους (head/tail) για να περιγράψουμε την αρχή και το τέλος και τους συνδέουμε.

```java
public CustomLinkedList() {  1 usage  nikosNikiforos
    head = new Node<>(null, null);
    tail = new Node<>(null, null);
    head.next = tail;//Σύνδεση
    tail.prev = head;//λίστας
    size = 0;
}
```

Στη συνέχεια υλοποιούμε τις βασικές μεθόδους της λίστας μας:

1. addFirst(Node<K, V> node) για την προσθήκη ενός κόμβου στην αρχή.

```
// Προσθήκη κόμβου στην αρχή
public void addFirst(Node<K, V> node) {  3 usages  nikosNikiforos
    node.next = head.next;
    node.prev = head;
    head.next.prev = node;
    head.next = node;
    size++;
```

2. remove(Node<K, V> node) για την αφαίρεση κόμβου από τη λίστα

```
// Αφαίρεση κόμβου από τη λίστα
public void remove(Node<K, V> node) {  3 usages  nikosNikiforos
    node.prev.next = node.next;
    node.next.prev = node.prev;
    size--;
```

3. removeLast() Που χρησιμοποιείται στην remove(Node<K,V>node) για την αφαίρεση του τελευταίου όμβου.

```
// Αφαίρεση του τελευταίου κόμβου
public Node<K, V> removeLast() {  1 usage  nikosNikiforos
    if (size == 0) return null;
    Node<K, V> lastNode = tail.prev;
    remove(lastNode);
    return lastNode;
}
```

4. getHead() getTail() size Είναι κάποιες διευκολυντικές μέθοδοι

για να έχουμε εύκολη πρόσβαση στην κορυφή, την ουρά και το μέγεθος της λίστας(αρχικά).

```java
    // Επιστροφή του πρώτου κόμβου
    public Node<K, V> getHead() {  1 usage  nikosNikiforos
        return size == 0 ? null : head.next;
    }

    // Επιστροφή του τελευταίου κόμβου
    public Node<K, V> getTail() {  1 usage  nikosNikiforos
        return size == 0 ? null : tail.prev;
    }

    // Επιστροφή μεγέθους της λίστας
    public int size() {  nikosNikiforos
        return size;
    }
}
```

<u>Υλοποίηση LRUCache:</u>

Η μνήμη μας συνδυάζει πίνακες κατακερματισμού και την διπλά συνδεδεμένη λίστα που υλοποιήσαμε για να ακολουθεί την στρατηγική LRU. Να αφαιρεί από τη μνήμη δηλαδή το στοιχείο που χρησιμοποιήσαμε πιο παλιά, όταν δεν υπάρχει χώρος, και να τοποθετεί στην αρχή το στοιχείο που χρησιμοποιήσαμε πιο πρόσφατα.

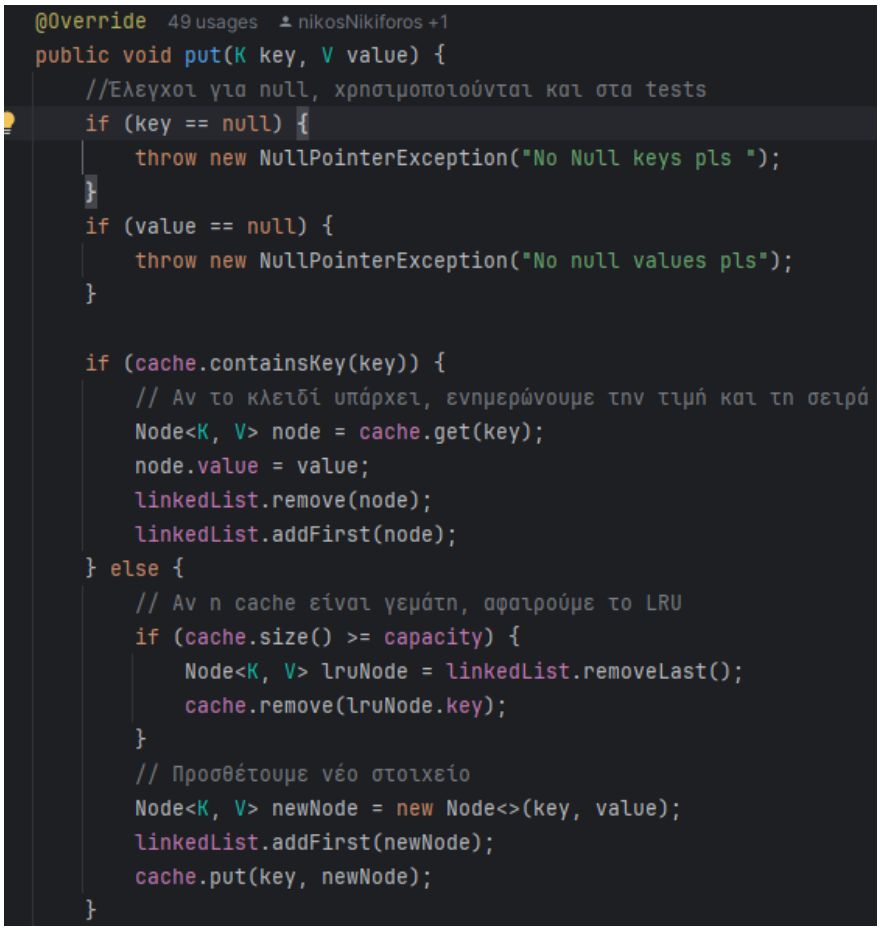
Η LRUCache είναι μια υλοποίηση της διεπαφής(interface) Cache το οποίο μας δόθηκε από την εκφώνηση.

```java
package org.hua.cache;
import java.util.HashMap;
import java.util.Map;


// Υλοποίηση της LRU
public class LRUCache<K,V> implements Cache<K,V> {  35 usages  nikosNikiforos +1
    private final int capacity;  2 usages
    private final Map<K, Node<K, V>> cache;  8 usages
    private final CustomLinkedList<K, V> linkedList;  10 usages
```

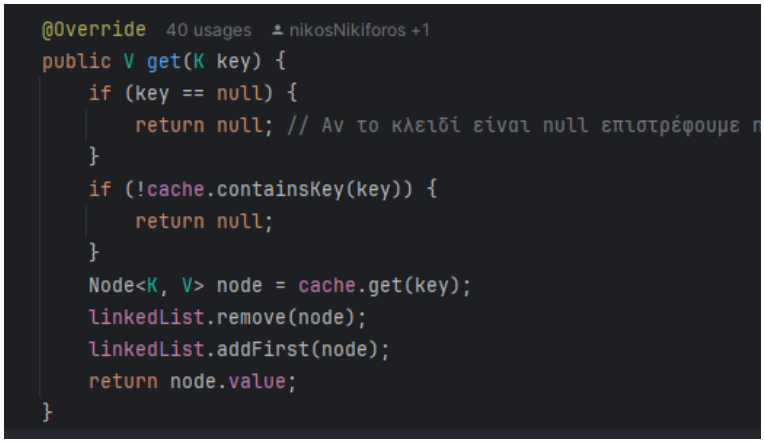
```java
package org.hua.cache;
//το interface όπως δόθηκε στην εκφώνηση
/**
 * A cache interface
 *
 * @param <K> the key type
 * @param <V> the value type
 */
public interface Cache<K, V> {  1 usage  1 implementation  nikosNikifor

    /**
     * Get the value for a key. Returns null if the key is not
     * in the cache.
     *
     * @param key the key
     */
    V get(K key);  40 usages  1 implementation  nikosNikiforos

    /**
     * Put a new key-value pair in the cache
     *
     * @param key the key
     * @param value the value
     */
    void put(K key, V value);  49 usages  1 implementation  nikosNikif
}
```

Οι μέθοδοι που υλοποιεί είναι οι απαραίτητες από το interface get/put και κάποιες βοηθητικές που προσθέσαμε εμείς.

1. put(K key, V value) Προσθέτει ένα στοιχείο ή ενημερώνει ένα υπάρχον.Αν η μνήμη είναι γεμάτη αφαιρεί το τελευταίο.



2. get(K key) Αν υπάρχει το κλειδί το επιστρέφει και το τοποθετεί στην αρχή και διαγράφει το τελευταίο. Για τιμές null στο κλειδί ή στην αξία επιστρέφει null.



3. getHead()/getTail() Επιστρέφουν την κορυφή και την ουρά της μνήμης μέσω των αντίστοιχων μεθόδων για την συνδεδεμένη λίστα.

```
public V getHead() {  11 usages  nikosNikiforos +1
    Node<K, V> headNode = linkedList.getHead();
    return headNode != null ? headNode.value : null;
}

public V getTail() {  9 usages  nikosNikiforos +1
    Node<K, V> tailNode = linkedList.getTail();
    return tailNode != null ? tailNode.value : null;
```

4. size() Μέσω της αντίστοιχης από την CustomLinkedList για να παίρνουμε εύκολα τον αριθμό των κόμβων μέσα στη λίστα.

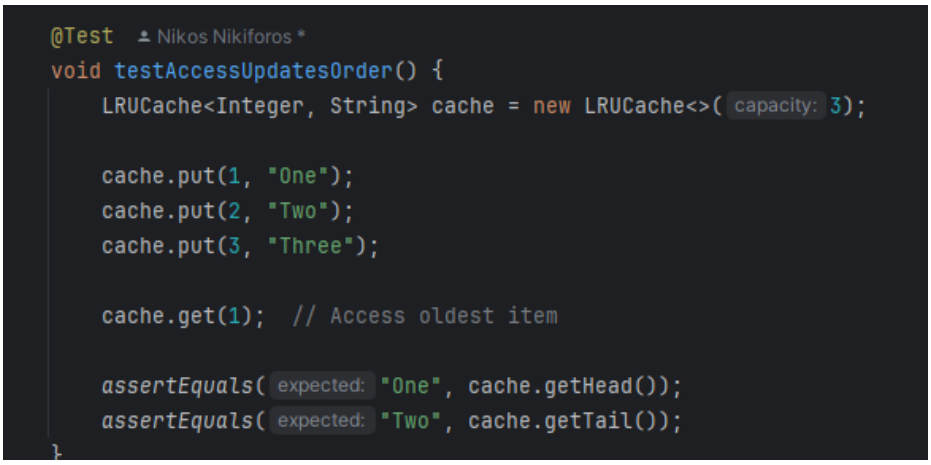
```
public int size() {  nikosNikiforos
    return linkedList.size();
}
```

Junit Tests

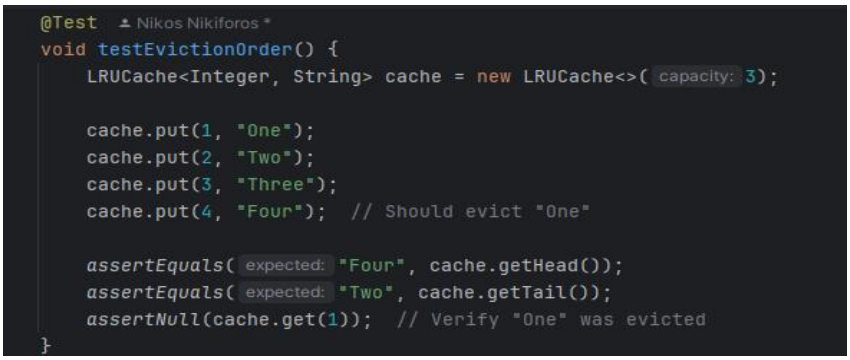
testHeadTailOrder: Αυτή η δοκιμή επαληθεύει τη βασική λειτουργικότητα της σειράς των στοιχείων στην κρυφή μνήμη. Ελέγχει αν τα στοιχεία τοποθετούνται σωστά, με το πιο πρόσφατα προστεθέν στοιχείο να βρίσκεται στην κορυφή (head) και το παλαιότερο στην ουρά (tail). Η δοκιμή είναι σημαντική καθώς επιβεβαιώνει την ορθή διατήρηση της χρονικής σειράς των στοιχείων.

```
@Test  Nikos Nikiforos *
void testHeadTailOrder() {
    LRUCache<Integer, String> cache = new LRUCache<>( capacity: 3);
    cache.put(1, "One");
    cache.put(2, "Two");
    cache.put(3, "Three");

    assertEquals( expected: "Three", cache.getHead());
    assertEquals( expected: "One", cache.getTail());
}
```

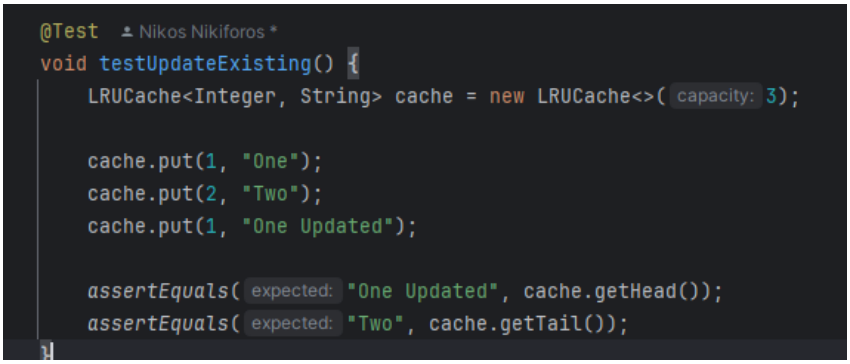
testAccessUpdatesOrder: Η δοκιμή αυτή εξετάζει αν η πρόσβαση σε ένα υπάρχον στοιχείο ενημερώνει σωστά τη σειρά των στοιχείων. Όταν προσπελαύνουμε ένα στοιχείο, αυτό πρέπει να μετακινείται στην κορυφή της κρυφής μνήμης, καθώς είναι το πιο πρόσφατα χρησιμοποιημένο.



testEvictionOrder: Εδώ ελέγχεται η διαδικασία απομάκρυνσης στοιχείων όταν η κρυφή μνήμη γεμίσει. Επιβεβαιώνει ότι όταν προστίθεται ένα νέο στοιχείο σε μια γεμάτη κρυφή μνήμη, το παλαιότερο στοιχείο (που βρίσκεται στην ουρά) απομακρύνεται σωστά.



testUpdateExisting: Το τέστ επαληθεύει τη συμπεριφορά της κρυφής μνήμης κατά την ενημέρωση υπάρχοντων στοιχείων. Όταν ενημερώνουμε ένα υπάρχον κλειδί, η τιμή του πρέπει να αλλάζει και το στοιχείο να μετακινείται στην κορυφή ως το πιο πρόσφατα χρησιμοποιημένο.



testUpdateExisting2: Αυτό το τέστ είναι extend του προηγούμενου, ελέγχοντας πιο διεξοδικά τη συμπεριφορά των ενημερώσεων. Επιβεβαιώνει ότι μετά από πολλαπλές ενημερώσεις και προσθήκες, η σειρά LRU διατηρείται σωστά και τα παλαιότερα στοιχεία απομακρύνονται .

```java
@Test  aimpapa *
void testUpdateExisting2() {

    LRUCache<Integer, String> cache = new LRUCache<>( capacity: 3);
    cache.put(1, "One");
    cache.put(2, "Two");
    cache.put(1, "One Updated");

    assertEquals( expected: "One Updated", cache.get(1));
    assertEquals( expected: "Two", cache.get(2));


    cache.put(3, "Three");
    cache.put(4, "Four");

    assertNull(cache.get(1));
    assertEquals( expected: "Two", cache.get(2));
    assertEquals( expected: "Three", cache.get(3));
    assertEquals( expected: "Four", cache.get(4));
}
```

testFrequentAccess: Αυτό το τέστ εξετάζει τη συμπεριφορά της κρυφής μνήμης σε συχνή πρόσβαση του ίδιου στοιχείου. Επαληθεύει ότι το συχνά προσπελαυνόμενο στοιχείο παραμένει στην κορυφή της κρυφής μνήμης.

```java
@Test  Nikos Nikiforos *
void testFrequentAccess() {
    LRUCache<Integer, String> cache = new LRUCache<>( capacity: 3);

    cache.put(1, "One");
    cache.put(2, "Two");
    cache.put(3, "Three");

    for (int i = 0; i < 5; i++) {
        cache.get(1);
        assertEquals( expected: "One", cache.getHead());
    }
}
```

testCapacityEnforcement: Αυτό το τέστ ελέγχει την χωρητικότητα της κρυφής μνήμης. Επιβεβαιώνει ότι η κρυφή μνήμη δεν υπερβαίνει ποτέ το καθορισμένο μέγεθός της, ακόμη και μετά από πολλαπλές προσθήκες στοιχείων.

```
@Test  Nikos Nikiforos +1 *
void testCapacityEnforcement() {
    LRUCache<Integer, String> cache = new LRUCache<>( capacity: 3);

    for (int i = 0; i < 10; i++) {
        cache.put(i, "Value" + i);
    }

    assertEquals( expected: 3, cache.size());
    assertEquals( expected: "Value9", cache.getHead());
}
```

testBasicFunctionality: Αυτή η δοκιμή ελέγχει τις βασικές λειτουργίες της κρυφής μνήμης σε ένα ενιαίο σενάριο. Συνδυάζει τις λειτουργίες προσθήκης, ανάκτησης και απομάκρυνσης στοιχείων, επιβεβαιώνοντας ότι όλα λειτουργούν σωστά μαζί.

```
@Test  aimpapa +1 *
void testBasicFunctionality() {

    LRUCache<Integer, String> cache = new LRUCache<>( capacity: 3);
    cache.put(1, "One");
    cache.put(2, "Two");
    cache.put(3, "Three");

    assertEquals( expected: "One", cache.get(1));
    assertEquals( expected: "Two", cache.get(2));
    assertEquals( expected: "Three", cache.get(3));

    cache.put(4, "Four");

    assertNull(cache.get(1),  message: "Item 1 should have been evicted");
    assertEquals( expected: "Two", cache.get(2));
    assertEquals( expected: "Three", cache.get(3));
    assertEquals( expected: "Four", cache.get(4));
} //aimilios telos
```

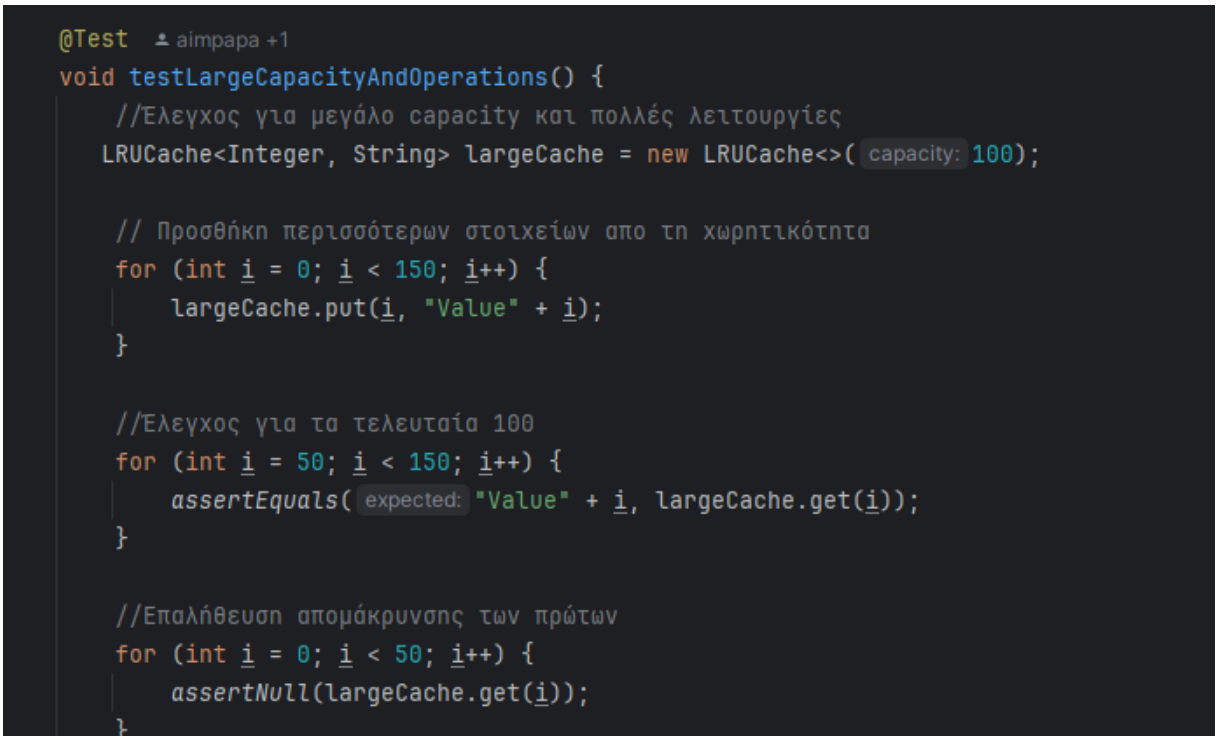
testSingleCapacity:Το τέστ εξετάζει τη συμπεριφορά της lru cach όταν έχει χωρητικότητα μόλις ενός στοιχείου. Πρόκειται για edge case και είναι σημαντικό για την επιβεβαίωση της ορθής λειτουργίας της cache σε ελάχιστο μέγεθος. Ελέγχει ότι κάθε νέα εισαγωγή διαγράφει το προηγούμενο στοιχείο και η πρόσβαση στο διαγραμμένο στοιχείο επιστρέφει null.

```
@Test  aimpapa *
void testSingleCapacity() {

    LRUCache<Integer, String> singleCache = new LRUCache<>( capacity: 1);

    singleCache.put(1, "One");
    assertEquals( expected: "One", singleCache.get(1));

    singleCache.put(2, "Two");
    assertNull(singleCache.get(1),  message: "First item should be evicted");
    assertEquals( expected: "Two", singleCache.get(2));
}
```

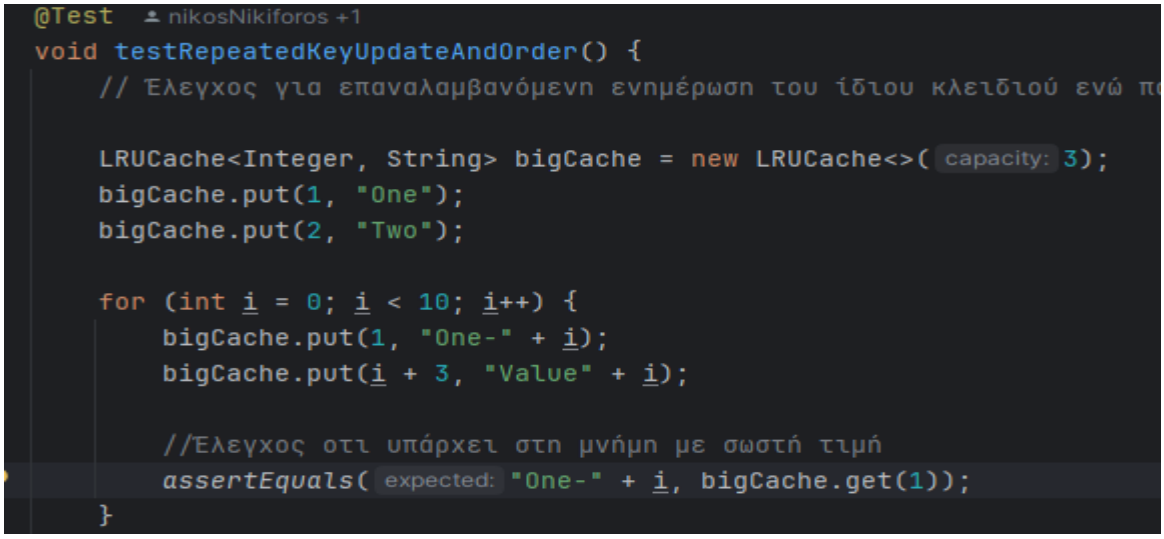
testAccessPattern: Το τέστ αυτό εξετάζει πώς τα διαφορετικά μοτίβα πρόσβασης στα στοιχεία επηρεάζουν τη θέση τους στην cache. Επαληθεύει ότι η πολιτική LRU λειτουργεί σωστά όταν προσπελαύνουμε στοιχεία με συγκεκριμένη σειρά, και ότι τα λιγότερο πρόσφατα χρησιμοποιημένα στοιχεία απομακρύνονται σωστά όταν η cache γεμίσει.

```
@Test  aimpapa *
void testAccessPattern() {

    LRUCache<Integer, String> cache = new LRUCache<>( capacity: 3);
    cache.put(1, "One");
    cache.put(2, "Two");
    cache.put(3, "Three");

    cache.get(1);
    cache.get(2);

    cache.put(4, "Four");

    assertNull(cache.get(3),  message: "Item 3 should be evicted");
    assertEquals( expected: "One", cache.get(1));
    assertEquals( expected: "Two", cache.get(2));
    assertEquals( expected: "Four", cache.get(4));
}
```

testLargeCapacity: Σε αυτή τη δοκιμή εξετάζουμε την συμπεριφορά της μνήμης σε μια περίπτωση με μεγάλο capacity. Στόχος είναι να γεμίσουμε και να ξεπεράσουμε την χωριτικότητα και στη συνέχεια να εξετάσουμε ότι έμειναν επιτυχώς τα τελευταία στοιχεία και απομακρύνθηκαν σωστά τα πρώτα.Στο παράδειγμα φτιάχνουμε μνήμη με χωριτικότητα 100, την γεμίζουμε με 150 στοιχεία και εξτάζουμε αν έμειναν τα τελευταία 100 και τα πρώτα 50 είναι εκτός(null).



testRepeatedKeyUpdateAndOrder: Έλεγχος για επαναλαμβανόμενη ενημέρωση του ίδιου κλειδιού την ώρα που προσθέτουμε με μια επανάληψη στοιχεία στη μνήμη. Ανανεώνουμε μονίμως με key 1 και προσθέτουμε με τη λούπα. Στο τέλος πρέπει το κλειδί 1 να υπάρχει με την σωστη τιμή.



testEmptyCache: Έλεγχος για συμπεριφορά κενής μνήμης. Δημιουργούμε μια μνήμη και ζητάμε από αυτή στοιχεία που δεν έχουν καταχωρηθεί. Πρέπει να μας επιστρέψει null.

```
@Test  nikosNikiforos +1
void testEmptyCache() {
    // Δοκιμή για κενή μνήμη
    LRUCache<Integer, String> cache = new LRUCache<>( capacity: 3);
    assertNull(cache.get(1),  message: "Empty cache so any key is null");
    assertNull(cache.get(0),  message: "Empty cache so any key is null");
}
```

testNullHandling: Έλεγχος για τη διαχείρηση για την απόπειρα καταχώρησης null κλειδιών και αξιών, καθώς και για το cache.get(null). Στη συνέχεια βάλαμε και δύο σωστές καταχωρήσεις για να δείξουμε οτι λειτουργεί κανονικά

```
@Test  nikosNikiforos +1
void testNullHandling() {
    LRUCache<Integer, String> cache = new LRUCache<>( capacity: 3);
    // Έλεγχος ότι το null key ρίχνει NullPointerException
    assertThrows(NullPointerException.class, () -> cache.put(null, "Value"));
    // Έλεγχος ότι το null value ρίχνει NullPointerException
    assertThrows(NullPointerException.class, () -> cache.put(1, null));
    // Έλεγχος ότι η get με null κλειδί επιστρέφει null
    assertNull(cache.get(null));

    // Κανονική εισαγωγή
    cache.put(1, "One");
    cache.put(2, "Two");
    assertEquals( expected: "One", cache.get(1)); // Ελέγχουμε την τιμή

    //Πρέπει να παραμείνει με 2 στοιχεία
    assertEquals( expected: 2, cache.size());
}
```

testMixedOperations: Μία συνθήκη με διάφορες λειτουργίες με στόχο να μπερδέψουμε τη μνήμη μας, όπου με τον ίδιο counter προσθέτουμε στοιχεία και τα ζητάμε μέσα από μια άλλη επανάληψη.

```
void testMixedOperations() {
    //Έλεγχος υπό διάφορα operations
    LRUCache<Integer, String> cache = new LRUCache<>( capacity: 3);
    for (int i = 0; i < 5; i++) {
        cache.put(i, "Value" + i);

        // Intermix gets with puts
        if (i > 0) {
            cache.get(i - 1);
        }
    }

    //
    assertNull(cache.get(0),  message: "First item should be out");
    assertNotNull(cache.get(4),  message: "Last item should be in cache");
    assertNotNull(cache.get(3),  message: "Recently accessed item should be in cache");
}
```

testRepeatedPutsOnSameKey: Σε αυτό το test εξετάζουμε την περίπτωση όπου τοποθετούμε πολλές φορές στη μνήμη με το ίδιο κλειδί αλλά διαφορετική αξία. Ελέγχουμε μετά το τέλος της επανάληψης την αναμενόμενη αξία και βάζουμε και δύο τιμές για να δείξουμε ότι θα παραμείνει στη μνήμη, εφόσον δεν ξεπερνάμε το capacity.

```
@Test  aimpapa +1
void testRepeatedPutsOnSameKey() {
    //Ελέγχει την επαναλαμβανόμενη τοποθέτηση στο ίδιο κλειδί

    LRUCache<Integer, String> cache = new LRUCache<>( capacity: 3);
    for (int i = 0; i < 100; i++) {
        cache.put(1, "Value" + i);
    }

    assertEquals( expected: "Value99", cache.get(1));

    // Fill remaining cache
    cache.put(2, "Two");
    cache.put(3, "Three");

    // Ελέγχει οτι το κλειδί 1 δεν βγήκε από τη μνήμη
    assertEquals( expected: "Value99", cache.get(1));
}
```

testHeadTailPositions: Ουσιαστικά σε αυτή τη δοκιμή εξετάζουμε την λειτουργικότητα των getHead() & getTail(). Αν επιστρέφει σωστά την κορυφή και την ουρά της λίστας μετά απο κάποιες εισχωρήσεις.

```
@Test  aimpapa +1
void testHeadTailPositions() {
    //Εξετάζουμε πως όντως παίρνουμε σωστά την κορυφή και την ουρά της μνήμης μέσω των getHead/Tail αντίστοιχα
    LRUCache<Integer, String> cache = new LRUCache<>( capacity: 3);
    // Αρχικός Έλεγχος
    assertNull(cache.getHead(),  message: "Head is null in empty cache");
    assertNull(cache.getTail(),  message: "Tail is null in empty cache");

    // Μοναδική εισαγωγή
    cache.put(1, "One");
    assertEquals( expe                                    ead should be the only item");
    assertEquals( expe                                    ail should be the only item");

    //Πολλαπλές εισαγωγές
    cache.put(2, "Two");
    cache.put(3, "Three");
    assertEquals( expected: "Three", cache.getHead(),  message: "Head should be most recent item");
    assertEquals( expected: "One", cache.getTail(),  message: "Tail should be oldest item");

    // Αφού πειράξουμε τη σειρά
    cache.get(1);  // Access oldest item
    assertEquals( expected: "One", cache.getHead(),  message: "Head should be recently accessed item");
    assertEquals( expected: "Two", cache.getTail(),  message: "Tail should be least recently used");

    // Έλεγχος νέας προσθήκης και διαγραφής LRU κόμβου
    cache.put(4, "Four");
    assertEquals( expected: "Four", cache.getHead(),  message: "Head should be new item");
    assertEquals( expected: "Three", cache.getTail(),  message: "Tail should be oldest remaining item");
    assertNull(cache.get(2),  message: "Kicked item should be null");
}
```

No documentation found.

Να σημειώσουμε κλείνοντας πως για να υλοποιήσουμε επιτυχώς τα junit tests στο IDE που χρησιμοποιήσαμε, χρειάστηκε να προσθέσουμε το απαραίτητο dependency στο pom.xml και να κάνουμε τα κατάλληλα imports.

```
    <dependency>
        <groupId>org.junit.jupiter</groupId>
        <artifactId>junit-jupiter</artifactId>
        <version>5.9.3</version>
        <scope>test</scope>
    </dependency>
</dependencies>
```

```
import org.hua.cache.LRUCache;
import org.junit.jupiter.api.Test;
import static org.junit.jupiter.api.Assertions.*;
```

2.1

Αρχιτεκτονική Σχεδίαση:

- Δημιουργήθηκε μια αφηρημένη κλάση (Abstract Class) για τη διαχείριση των μετρητών hits/misses, η οποία υλοποιεί το βασικό interface Counter που δημιουργησαμε.

Class MyCache:

- Η LRUCache class μετονομάστηκε σε MyCache
- Υλοποιήθηκαν δύο κατασκευαστές:
    - Ένας προεπιλεγμένος κατασκευαστής για βασική λειτουργικότητα
    - Ένας εξειδικευμένος κατασκευαστής που δέχεται παράμετρο τύπου CacheReplacementPolicy, επιτρέποντας την επιλογή στρατηγικής (LRU ή MRU)

- Η μέθοδος put τροποποιήθηκε ώστε να περιλαμβάνει μια συνθήκη ελέγχου που καθορίζει την κατάλληλη στρατηγική αντικατάστασης. Αυτό επιτρέπει στο σύστημα να επιλέγει δυναμικά μεταξύ LRU και MRU, διατηρώντας παράλληλα την αναγνωσιμότητα του κώδικα.

```
@Override  51 usages  nikosNikiforos
public void put(K key, V value) {
    if (key == null || value == null) {
        throw new NullPointerException("Key and value cannot be null.");
    }

    if (cache.containsKey(key)) {
        // Αν το κλειδί υπάρχει ενημερώνουμε την τιμή και τη θέση
        Node<K, V> node = cache.get(key);
        node.value = value;
        linkedList.remove(node);
        linkedList.addFirst(node); // Μεταφορά στην αρχή για LRU
    } else {
        if (cache.size() >= capacity) {
            Node<K, V> nodeToRemove;
            // Επιλογή στρατηγικής αντικατάστασης
            if (policy == CacheReplacementPolicy.LRU) {
                nodeToRemove = linkedList.removeLast(); // Αφαίρεση LRU
            } else { // MRU
                nodeToRemove = linkedList.removeFirst(); // Αφαίρεση MRU
            }
            cache.remove(nodeToRemove.key);
        }
        // Προσθήκη νέου στοιχείου
        Node<K, V> newNode = new Node<>(key, value);
        linkedList.addFirst(newNode);
        cache.put(key, newNode);
    }
}
```

<u>Η main :</u>

- Στη main μέθοδο, υλοποιήθηκαν τρία διαφορετικά σενάρια δοκιμών με διαφορετικές χωρητικότητες cache (10, 20, 40), επιτρέποντας τη συγκριτική ανάλυση της απόδοσης των στρατηγικών LRU και MRU.
- Δημιουργήσαμε ένα πίνακα με τις 3 διαφορετικές χωρητικότητες: int[] cacheSizes = {10, 20, 40};
- Χρησιμοποιήσαμε μια μέθοδο testPolicy() η οποία μας δίνει την δυνατότητα να κάνουμε δοκιμές με διαφορετικές χωρητικότητες και στρατηγικές για τη μνήμη που δημιουργήσαμε.Επίσης έχει και ένα σενάριο με τα απαραίτητα operations για να εξετάσουμε την λειτουργία της κάθε μνήμης.Μέσα στο σενάριο αυτό χρησιμοποιήσαμε και μια τυχαία δεκαδική τιμή η οποία εφαρμόζει ουσιαστικά την 80/20 κατανομή.Τέλος εμφανίζει τα αποτελέσματα στη κονσόλα εκτυπώνοντας τις επιτυχίες, τις αποτυχίες σε απόλυτο αριθμό αλλά και σε ποσοστά.

```
private static void testPolicy(int capacity, CacheReplacementPolicy policy ) {
    System.out.println("Testing " + policy.getDescription());

    // Δημιουργία cache με συγκεκριμένη στρατηγική και χωρητικότητα
    MyCache<Integer, String> cache = new MyCache<>(capacity, policy);
    Random random = new Random();
    int operations = 100000;

    for (int i = 0; i < operations; i++) {
        int key;
        if (random.nextDouble() < 0.8) {
            key = random.nextInt( bound: 20); // 80% αιτήματα για hot keys (0-
        } else {
            key = random.nextInt( bound: 80) + 20; // 20% αιτήματα για cold ke
        }

        // Ανάκτηση ή εισαγωγή στοιχείου
        String value = cache.get(key);
        if (value == null) {
            cache.put(key, "Value" + key);
        }
    }

    // Εκτύπωση αποτελεσμάτων
    System.out.println("Total operations: " + operations);
    System.out.println("Cache Hits: " + cache.getHitCount());
    System.out.println("Cache Misses: " + cache.getMissCount());
    System.out.printf("Hit Rate: %.2f%%\n", cache.getHitRate());
    System.out.printf("Miss Rate: %.2f%%\n", cache.getMissRate());
}
```

- Τέλος χρησιμοποιούμε μια επανάληψη για τις τιμές των cacheSizes ώστε να κληθεί η testPolicy με διαφορετική χωρητικότητα και στρατηγική κάθε φορά.

```
    // Διαφορετικά μεγέθη cache για ανάλυση απόδοσης
    int[] cacheSizes = {10, 20, 40};

    System.out.println("Cache Performance Analysis");
    System.out.println("=========================\n");

    for (int capacity : cacheSizes) {
        System.out.println("Cache Capacity: " + capacity);
        System.out.println("----------------");

        // Δοκιμή LRU
        System.out.println("\nLRU (Least Recently Used) Strategy:");
        testPolicy(capacity , CacheReplacementPolicy.LRU);

        // Δοκιμή MRU
        System.out.println("\nMRU (Most Recently Used) Strategy:");
        testPolicy(capacity, CacheReplacementPolicy.MRU);

        System.out.println("\n=========================\n");
    }
}
```

Αποτελέσματα Main και σύγκριση :
Για την κατανόηση της επίδρασης του μεγέθους της cache στην απόδοση, εξετάστηκαν τρεις διαφορετικές χωρητικότητες (10, 20 και 40 καταχωρήσεις). Το σενάριο δοκιμής περιελάμβανε 100.000 λειτουργίες με κατανομή πρόσβασης 80/20, που σημαίνει ότι το 80% των αιτημάτων στόχευε στο 20% των δεδομένων.
Τα αποτελέσματα δείχνουν ότι το μέγεθος της Cache επηρεάζει σημαντικά την απόδοση. Στην LRU η αύξηση της χωρητικότητας από 10 σε 40 βελτίωσε δραματικά το ποσοστό επιτυχίας:

```
Cache Capacity: 10
---------------

LRU (Least Recently Used) Strategy:
Testing Least Recently Used
Total operations: 100000
Cache Hits: 31237
Cache Misses: 68763
Hit Rate: 31.24%
Miss Rate: 68.76%
```

```
Cache Capacity: 20
---------------

LRU (Least Recently Used) Strategy:
Testing Least Recently Used
Total operations: 100000
Cache Hits: 58269
Cache Misses: 41731
Hit Rate: 58.27%
Miss Rate: 41.73%
```

```
Cache Capacity: 40
----------------

LRU (Least Recently Used) Strategy:
Testing Least Recently Used
Total operations: 100000
Cache Hits: 84069
Cache Misses: 15931
Hit Rate: 84.07%
Miss Rate: 15.93%
```

Η βελτίωση αυτή είναι λογική, καθώς μια μεγαλύτερη cache μπορεί να αποθηκεύσει περισσότερα από τα συχνά προσπελαυνόμενα στοιχεία. Η αύξηση από 31,24% σε 84,07% στο ποσοστό επιτυχίας καταδεικνύει πόσο κρίσιμη είναι η σωστή χωρητικότητα της cache για την απόδοση.
Σε γενικό βαθμό παρατηρούμε πως ενώ η LRU επωφελείται σταθερά από την αύξηση της χωρητικότητας, με ένα κρίσιμο και επαρκές σημείο τις 40 καταχωρήσεις, η MRU κρατάει μικρά ποσοστά συνέχεια.

```
Cache Capacity: 10
---------------

LRU (Least Recently Used) Strategy:
Testing Least Recently Used
Total operations: 100000
Cache Hits: 31237
Cache Misses: 68763
Hit Rate: 31.24%
Miss Rate: 68.76%

MRU (Most Recently Used) Strategy:
Testing Most Recently Used
Total operations: 100000
Cache Hits: 12109
Cache Misses: 87891
Hit Rate: 12.11%
Miss Rate: 87.89%
```

```
Cache Capacity: 20
---------------

LRU (Least Recently Used) Strategy:
Testing Least Recently Used
Total operations: 100000
Cache Hits: 58269
Cache Misses: 41731
Hit Rate: 58.27%
Miss Rate: 41.73%

MRU (Most Recently Used) Strategy:
Testing Most Recently Used
Total operations: 100000
Cache Hits: 22113
Cache Misses: 77887
Hit Rate: 22.11%
Miss Rate: 77.89%
```

```
Cache Capacity: 40
---------------

LRU (Least Recently Used) Strategy:
Testing Least Recently Used
Total operations: 100000
Cache Hits: 84069
Cache Misses: 15931
Hit Rate: 84.07%
Miss Rate: 15.93%

MRU (Most Recently Used) Strategy:
Testing Most Recently Used
Total operations: 100000
Cache Hits: 41846
Cache Misses: 58154
Hit Rate: 41.85%
Miss Rate: 58.15%
```

Αυτή η σημαντική διαφορά εξηγείται από τον τρόπο λειτουργίας κάθε πολιτικής σε σχέση με τον τύπο πρόσβασης. Η LRU διατηρεί τα συχνά χρησιμοποιούμενα στοιχεία στην cache, συμβαδίζοντας με την κατανομή 80/20 όπου τα περισσότερα αιτήματα κατευθύνονται σε ένα μικρό σύνολο στοιχείων. Αντίθετα, η MRU αφαιρεί τα πιο πρόσφατα χρησιμοποιημένα στοιχεία, τα οποία παραδόξως είναι πιθανότερο να χρειαστούν ξανά σύντομα.

Παρατηρήσεις :

- Οι δοκιμές δείχνουν πως για την LRU την μέγιστη απόδοση, με σημαντική διαφορά, πετυχαίνουμε με χωρητικότητα 40 καταχωρήσεις. Καταχωρήσεις διπλάσιες δηλαδή από τα κλειδιά που επισκεπτώμαστε το 80% των περιπτώσεων, πράγμα που σημαίνει πως ακόμα με τέτοια συχνότητα επισκεψιμότητας σε κλειδιά πάλι χρειαζόμαστε επαρκή χώρο για τις υπόλοιπες περιπτώσεις.
- Η LRU είναι σαφώς η ανώτερη επιλογή για εργασίες με επαναλλαμβανόμενη επισκεψιμότητα σε κλειδιά, εφόσον κρατάει τα στοιχεία που χρησιμοποιήθηκαν πιο πρόσφατα.
- Η MRU δεν επωφελείται στο συγκεκριμένο σενάριο από την αύξηση χωρητικότητας των καταχωρήσεων επειδή αφαιρεί άμεσα τα στοιχεία που χρησιμοποιούμε 8 στις 10 φορές.
- Η επιλογή της Cache πρέπει να γίνεται προσεκτικά καθώς επηρεάζει την απόδοση αρκετά, με ισσοροπία όμως ώστε να μην γίνεται και σπατάλη πόρων. Παράδειγμα με Capacity 60.

```
Cache Capacity: 60
----------------

LRU (Least Recently Used) Strategy:
Testing Least Recently Used
Total operations: 100000
Cache Hits: 89841
Cache Misses: 10159
Hit Rate: 89.84%
Miss Rate: 10.16%
```

Δεν κερδίζουμε ιδιαίτερα σε σχέση με μια LRU 40 καταχωρήσεων.

UNIT TEST:

- Τα unit test τροποποιήθηκαν απο τα αρχικά και υιοθετήθηκε η προσέγγιση @ParameterizedTest με χρήση του @EnumSource , λόγο της ύπαρξης του CacheReplacementPolicy enum και οτι στο part 3 θα προστεθεί αλλη μια cache μέθοδος.Έτσι μπορούμε με τις κατάλληλες παραμέτρους να ελέγξουμε μια ή παραπάνω Cache στα tests.

```java
@ParameterizedTest  aimpapa
@EnumSource(CacheReplacementPolicy.class)
void testAccessPattern(CacheReplacementPolicy policy) {
    // Έλεγχος μοτίβου πρόσβασης και επίδρασης στη σειρά LRU
    MyCache<Integer, String> cache = new MyCache<>( capacity: 3, policy);
    cache.put(1, "One");
    cache.put(2, "Two");
    cache.put(3, "Three");

    cache.get(1);
    cache.get(2);

    cache.put(4, "Four");

    if (policy == CacheReplacementPolicy.LRU) {
        assertNull(cache.get(3));
        assertEquals( expected: "One", cache.get(1));
        assertEquals( expected: "Two", cache.get(2));
    } else {
        assertNull(cache.get(2));
        assertEquals( expected: "Three", cache.get(3));
        assertEquals( expected: "One", cache.get(1));
    }
    assertEquals( expected: "Four", cache.get(4));
}
```

```
[INFO] Tests run: 34, Failures: 0, Errors: 0, Skipped: 0, Time elapsed: 0.141 s -- in MyCacheTest
[INFO]
[INFO] Results:
[INFO]
[INFO] Tests run: 34, Failures: 0, Errors: 0, Skipped: 0
[INFO]
[INFO] ------------------------------------------------------------------------
[INFO] BUILD SUCCESS
[INFO] ------------------------------------------------------------------------
[INFO] Total time:  2.179 s
[INFO] Finished at: 2025-01-10T03:19:43+02:00
[INFO] ------------------------------------------------------------------------
```