

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 είναι 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 x
1 package org.hua.cache;
2
3 public class Node<K, V> {
4     K key;           // Το κλειδί του κόμβου
5     V value;         // Η τιμή του κόμβου
6     Node<K, V> prev; // Δείκτης στον προηγούμενο κόμβο
7     Node<K, V> next; // Δείκτης στον επόμενο κόμβο
8
9     public Node(K key, V value) {
10         this.key = key;
11         this.value = value;
12     }
13 }
```

Υλοποίηση διπλά συνδεδεμένης λίστας:

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

```
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).

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

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

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

Υλοποίηση LRUCache:

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.

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

// Υλοποίηση της LRU
public class LRUCache<K, V> implements Cache<K, V> {
    private final int capacity;
    private final Map<K, Node<K, V>> cache;
    private final CustomLinkedList<K, V> linkedList;
}

package org.hua.cache;
//to interface όπως δόθηκε στην εκφώνηση
/**
 * A cache interface
 *
 * @param <K> the key type
 * @param <V> the value type
 */
public interface Cache<K, V> {
    /**
     * Get the value for a key. Returns null if the key is not
     * in the cache.
     *
     * @param key the key
     */
    V get(K key);

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

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.

```
@Override 49 usages  nikosNikiforos +1
public void put(K key, V value) {
    //Έλεγχοι για null, χρησιμοποιούνται και στα tests
    if (key == null) {
        throw new NullPointerException("No Null keys pls ");
    }
    if (value == null) {
        throw new NullPointerException("No null values pls");
    }

    if (cache.containsKey(key)) {
        // Αν το κλειδί υπάρχει, ενημερώνουμε την τιμή και τη σειρά
        Node<K, V> node = cache.get(key);
        node.value = value;
        linkedList.remove(node);
        linkedList.addFirst(node);
    } else {
        // Αν η cache είναι γεμάτη, αφαιρούμε το LRU
        if (cache.size() >= capacity) {
            Node<K, V> lruNode = linkedList.removeLast();
            cache.remove(lruNode.key);
        }
        // Προσθέτουμε νέο στοιχείο
        Node<K, V> newNode = new Node<>(key, value);
        linkedList.addFirst(newNode);
        cache.put(key, newNode);
    }
}
```

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.

```
@Override 40 usages  nikosNikiforos +1
public V get(K key) {
    if (key == null) {
        return null; // Αν το κλειδί είναι null επιστρέφουμε null
    }
    if (!cache.containsKey(key)) {
        return null;
    }
    Node<K, V> node = cache.get(key);
    linkedList.remove(node);
    linkedList.addFirst(node);
    return node.value;
}
```

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 προσπελάζουμε ένα στοιχείο, αυτό πρέπει να μετακινείται στην κορυφή της κρυφής μνήμης, καθώς είναι το πιο πρόσφατα χρησιμοποιημένο.

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

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

    cache.get(1); // Access oldest item

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

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.

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

    cache.put(1, "One");
    cache.put(2, "Two");
    cache.put(3, "Three");
    cache.put(4, "Four"); // Should evict "One"

    assertEquals( expected: "Four", cache.getHead());
    assertEquals( expected: "Two", cache.getTail());
    assertNull(cache.get(1)); // Verify "One" was evicted
}
```

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.

```
@Test  Nikos Nikiforos *
void testUpdateExisting() {
    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.getHead());
    assertEquals( expected: "Two", cache.getTail());
}
```

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 .

```
@Test
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.

```
@Test
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  ± aimppa +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 τρέστ 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  @aimapa *
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 τρέστ αυτό εξετάζει πώς τα διαφορετικά μοτίβα πρόσβασης στα στοιχεία επηρεάζουν τη θέση τους στην cache. Επαληθεύει ότι η πολιτική LRU λειτουργεί σωστά όταν προσπελάζουμε στοιχεία με συγκεκριμένη σειρά, και ότι τα λιγότερο πρόσφατα χρησιμοποιημένα στοιχεία απομακρύνονται σωστά όταν η cache γεμίσει.

```
@Test  @aimapa *
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 χωρητικότητα και στη συνέχεια να εξετάσουμε ότι έμειναν επιτυχώς τα τελευταία στοιχεία και απομακρύνθηκαν σωστά τα πρώτα. Στο παράδειγμα φτιάχνουμε μνήμη με χωρητικότητα 100, την γεμίζουμε με 150 στοιχεία και εξετάζουμε αν έμειναν τα τελευταία 100 και τα πρώτα 50 είναι εκτός(null).

```
@Test
void testLargeCapacityAndOperations() {
    //Έλεγχος για μεγάλο capacity και πολλές λειτουργίες
    LRUCache<Integer, String> largeCache = new LRUCache<>( capacity: 100);

    // Προσθήκη περισσότερων στοιχείων απο τη χωρητικότητα
    for (int i = 0; i < 150; i++) {
        largeCache.put(i, "Value" + i);
    }

    //Έλεγχος για τα τελευταία 100
    for (int i = 50; i < 150; i++) {
        assertEquals( expected: "Value" + i, largeCache.get(i));
    }

    //Επαλήθευση απομάκρυνσης των πρώτων
    for (int i = 0; i < 50; i++) {
        assertNull(largeCache.get(i));
    }
}
```

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 λούππα. Στο τέλος πρέπει το κλειδί 1 να υπάρχει με την σωστή τιμή.

```
@Test
void testRepeatedKeyUpdateAndOrder() {
    // Έλεγχος για επαναλαμβανόμενη ενημέρωση του ίδιου κλειδιού ενώ π

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

    for (int i = 0; i < 10; i++) {
        bigCache.put(1, "One-" + i);
        bigCache.put(i + 3, "Value" + i);
    }

    //Έλεγχος οτι υπάρχει στη μνήμη με σωστή τιμή
    assertEquals( expected: "One-" + i, bigCache.get(1));
}
```

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

```
@Test
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 διαχείριση 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 ότι λειτουργεί κανονικά

```
@Test
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.

```
@Test
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
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 ότι 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
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(expected: "One", cache.getHead(), message: "Head should be the only item");
    assertEquals(expected: "One", cache.getTail(), message: "Tail 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");
}
```

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

ΕΥΧΑΡΙΣΤΟΥΜΕ.