Test Suite & Test Cases di ListAdapter.java e MapAdapter.java

Introduzione

Questa suite vuole testare tutti i metodi presenti nelle due classi ListAdapter.java e MapAdapter.java presenti all'interno della directory. I test vengo effettuati approfonditamente per testare a fondo i metodi realizzati, i test descriveranno il comportamento delle funzioni sia nel caso di parametri passati correttamente sia nel caso in cui il valore del parametro passato non sia accettabile dal metodo, verranno descritti i casi in cui verranno lanciate le eccezioni. I test vengono eseguiti secondo l'ordine che viene utilizzato nella realizzazione nei file sorgente (alfabetico).

Nel caso dei test riguardanti le sottoclassi presenti è stato preferito sviluppare un unico metodo in cui sono stati ripresi gli stessi test fatti per il resto della classe, nel caso in cui questi test dovessero operare negli stessi modi, e così, avendoli già testati è bastato richiamare la stessa funzione per testare ma su un oggetto diverso (creato appositamente per la sottoclasse); per i metodi che sono specificatamente nuovi per una determinata sottoclasse invece, si è proceduto testando accuratamente il metodo nuovo. [Ad esempio, nella sublist il test del metodo add è identico a quello testato per la lista, solamente richiamato sull'oggetto "sub" creato nel test della sublist].

In questo documento andrò a spiegare brevemente il **funzionamento dei test** dei due adattatori che sono stati realizzati per l'Homework. Tutti i test verranno eseguiti direttamente dalla classe testRunner che riporta l'esito, il tempo, il numero dei test.

Nella pagina successiva seguono i test effettuati per la prima classe.

ListTest.java – verranno istanziati 3 oggetti ed un ListAdapter che potremmo usare dove richiesto nei test case

Di default il ListAdapter viene inizializzato vuoto

1. Test metodo list_add()
   - Sommario: si procede con l'aggiunta di alcuni elementi all'interno della lista per poi verificarne il corretto posizionamento
   - Test: public boolean add(Object o), boolean add(int index, Object o)
   - Design & Description: il test deve verificare il corretto inserimento degli elementi all'interno della lista nell'indice corretto se specificato, altrimenti aggiunge in coda
   - Pre-condizioni: metodo get() funzionante
   - Post-condizioni: la lista deve tornare ad essere vuota
     Risultato atteso: la lista conterrà un numero di elementi maggiore ogni volta che viene invocato un metodo add() e andrà ad incrementare la dimensione. Nel caso in cui l'oggetto passato sia *null* o l'indice non sia valido vengono lanciate *NullPointerException* o *IndexOutOfBoundsException*

2. Test metodo list_addAll()
   - Sommario: si procede con l'aggiunta di alcuni elementi all'interno della lista con una HCollection creata sul momento per poi verificare il corretto posizionamento degli elementi all'interno
   - Test: public boolean addAll(HCollection c), public boolean addAll(int index, HCollection c)
   - Design & Description: il test deve verificare il corretto inserimento degli elementi della collection nella lista, nell'indice corretto se specificato altrimenti aggiunge in coda
   - Pre-condizioni: metodo get() funzionante
   - Post-condizioni: la lista deve tornare ad essere vuota
   - Risultato atteso: la lista conterrà un numero di elementi maggiore ogni volta che viene eseguito un metodo addAll() ed andrà ad aumentare la dimensione della lista. Nel caso in cui la HCollection passata sia *null* o l'indice non sia valido vengono lanciate *NullPointerException* o *IndexOutOfBoundsException*

3. Test metodo list_clear()
   - Sommario: si procede con l'aggiunta di un elemento per poi rimuoverlo con il clear per poi verificare che la dimensione della lista risulti effettivamente 0
   - Test: public void clear()
   - Design & Description: il test deve verificare che la lista diventi vuota dopo la rimozione degli elementi aggiunti in precedenza
   - Pre-condizioni: metodo size() funzionante
   - Post-condizioni: la lista deve tornare ad essere vuota
   - Risultato atteso: la lista non conterrà più alcun elemento

4. Test metodo list_contains()
    - Sommario: si procede con l'aggiunta di due elementi, verifico poi che uno di questi due elementi sia contenuto mentre ipotizzo il falso affermando che ce ne sia un terzo che non ho mai aggiunto
    - Test: public boolean contains(Object o)
    - Design & Description: il test deve verificare che nella lista sia contenuto l'oggetto passato come paramentro, aggiunto precedentemente
    - Pre-condizioni: metodo add() funzionante
    - Post-condizioni: la lista deve tornare ad essere vuota
    - Risultato atteso: il metodo ritornerà il valore true se l'oggetto è presente all'interno della lista. Nel caso in cui l'oggetto passato sia *null* viene lanciata l'eccezione *NullPointerException*

5. Test metodo list_containsAll()
    - Sommario: aggiungo alcuni elementi alla lista, poi aggiungo alcuni di questi anche ad un'altra lista e verifico che quella più grande contenga tutti gli elementi della minore. Costruisco poi una lista in cui è presente almeno un elemento non presente nella lista grande e verifico che l'invocazione del metodo non dia esito positivo
    - Test: public boolean containsAll(HCollection c)
    - Design & Description: il test deve verificare che all'interno della lista siano presenti tutti gli elementi che sono presenti nella HCollection passata come paramentro
    - Pre-condizioni: Il metodo add() deve essere funzionante (sia per list che per HCollection)
    - Post-condizioni: la lista deve tornare ad essere vuota
    - Risultato atteso: il metodo deve ritornare true se gli elementi della HCollection sono tutti presenti nella lista. Nel caso in cui la HCollection passata sia *null* viene lanciata l'eccezione *NullPointerException*

6. Test metodo list_equals()
    - Sommario: procedo con l'aggiunta di due elementi alla lista e poi aggiungo gli stessi all'interno di un'altra appena creata, verifico poi che le due liste siano uguali
    - Test: public boolean equals(Object o)
    - Design & Description: Il test deve verificare che i due oggetti aggiunti in precedenza siano uguali
    - Pre-condizioni: Il metodo add() deve essere funzionante
    - Post-condizioni: la lista deve tornare ad essere vuota
    - Risultato atteso: il metodo deve ritornare true se i due oggetti sono uguali. Negli altri casi non viene lanciata eccezione ma viene restituito false

7. Test metodo list_get()
    - Sommario: si procede con l'aggiunta di due elementi nella lista per poi verificare con il metodo get() che questi siano nella posizione corretta in cui li ho inseriti
    - Test: public Object get(int index)
    - Design & Description: il test deve verificare che l'elemento inserito sia nella posizione corretta, confrontando l'indice di inserimento all'indice in cui è stato inserito nella lista
    - Pre-condizioni: il metodo add() deve essere funzionante per poter aggiungere gli elementi nella lista
    - Post-condizioni: la lista deve tornare ad essere vuota
    - Risultato atteso: Il metodo deve ritornare l'oggetto se presente all'indice della lista. Nel caso in cui l'indice non sia valido viene lanciata l'eccezione *IndexOutOfBoundException*

8. Test metodo list_indexOf()
    - Sommario: si procede con l'aggiunta di due elementi nella lista per poi verificare con il metodo indexOf() che la posizione in cui sono stati aggiunti sia corretta
    - Test: public int indexOf(Object o)
    - Design & Description: il test deve verificare che all'indice passato come parametro corrisponda l'oggetto precedentemente aggiunto
    - Pre-condizioni: il metodo add() deve essere funzionante per poter aggiungere gli elementi nella lista
    - Post-condizioni: la lista deve tornare ad essere vuota
    - Risultato atteso: Il metodo deve ritornare l'indice dell'oggetto passato se presente. Nel caso in cui l'oggetto sia *null* viene lanciata l'eccezione *NullPointerException*

9. Test metodo list_isEmpty()
    - Sommario: Si verifica che la lista sia vuota dopo aver aggiunto e successivamente tolto (con clear) un elemento
    - Test: public boolean isEmpty()
    - Design & Description: Il test deve verificare che la dimensione della lista sia priva di elementi al suo interno
    - Pre-condizioni:
    - Post-condizioni: la lista deve tornare ad essere vuota
    - Risultato atteso: Il metodo restituisce true se la lista risulta priva di elementi, false altrimenti.

10. Test metodo list_Iterator()

- Sommario: si crea un nuovo iteratore semplice ed un array di oggetti, aggiungo alla mappa degli oggetti e poi tramite l'iteratore della mappa li aggiungo nell'array. Verifico che l'array di elementi coincida con un nuovo array di oggetti creato sul momento con gli stessi oggetti, verifico che dopo un remove l'iteratore abbia comunque un elemento successivo, e che la dimensione si sia ridotta
- Test: public HIterator iterator(), hasNext(), next() e remove() dell'iteratore creato
- Design & Description: il test deve verificare che venga creato un iteratore semplice che opera sugli elementi della lista
- Pre-condizioni:
- Post-condizioni: la lista deve tornare ad essere vuota
- Risultato atteso: l'iteratore deve scorrere tanti elementi quanti sono quelli della lista, nell'ordine corretto. hasNext ritorna true solamente se una chiamata a next ritorna un elemento valido. Il metodo ritorna dunque un iteratore valido

11. Test metodo list_lastIndexOf()
    - Sommario: si procede con l'aggiunta di due elementi nella lista per poi verificare con il metodo lastIndexOf() che la posizione in cui sono stati aggiunti sia corretta
    - Test: public int lastIndexOf(Object o)
    - Design & Description: il test deve ritornare l'indice dell'ultimo elemento presente nella lista, passato come paremetro
    - Pre-condizioni: metodo add() funzionante
    - Post-condizioni: la lista deve tornare ad essere vuota
    - Risultato atteso: il metodo ritorna l'indice dell'ultimo elemento presente nella lista passato come parametro. Nel caso in cui l'oggetto sia *null* viene lanciata l'eccezione *NullPointerException*

12. Test metodo list_listIterator()
    - Sommario: si crea un nuovo listIterator, aggiungo elementi alla lista, verifico la dimensione e verifico che un determinato oggetto è stato aggiunto ad una determinata posizione nella lista con il listIterator. Verifico che l'iteratore abbia un precedente, verifico che l'elemento che mi aspetto sia il precedente e, dopo aver rimosso un oggetto con l'iteratore, controllo che la dimensione della lista sia diminuita di conseguenza.
    - Test: public HListIterator listIterator(int index), next(), hasNext(), previous(), hasPrevious() e remove
    - Design & Description: il test deve verificare che venga creato un iteratore capace di operare sugli elementi della lista in entrambi i versi di percorrenza
    - Pre-condizioni: il metodo add() deve essere funzionante per poter aggiungere gli elementi nella lista
    - Post-condizioni: la lista deve tornare ad essere vuota

- Risultato atteso: l'iteratore deve scorrere tanti elementi quanti sono quelli della lista, nell'ordine corretto. hasNext ritorna true solamente se una chiamata a next ritorna un elemento valido. Dopo aver iterato la lista hasNext ritorna false e next() lancia eccezione (uguale per hasPrevious() e previous()). Nel caso in cui l'indice passato a listIterator non sia valido viene lanciata l'eccezione *IndexOutOfBoundsException*

13. Test metodo list_remove()
    - Sommario: si procede aggiungendo elementi alla lista in un determinato ordine, per poi rimuovere degli elementi verificando di rimuovere l'oggetto che ci si aspetta. Si verifica poi di poter rimuovere altri elementi che sono presenti nella lista, si verifica il cambiamento della dimensione e provo ad affermare il falso tentando a rimuovere un elemento inizializzato ma non presente nella lista
    - Test: public Object remove(int index), public boolean remove(Object o)
    - Design & Description: il test deve verificare che entrambi i metodi rimuovano l'elemento posto all'indice passato oppure il primo oggetto uguale a quello passato, se è presente nella lista
    - Pre-condizioni: il metodo add() deve essere funzionante per poter aggiungere gli elementi nella lista
    - Post-condizioni: la lista deve tornare ad essere vuota
    - Risultato atteso: La lista deve essere vuota o con elementi ad esclusione da quello che è stato rimosso dalla chiamata del remove. Nel caso in cui l'indice non sia valido o l'oggetto sia *null* viene lanciata l'eccezione *IndexOutOfBoundException* o *NullPointerException*.

14. Test metodo list_removeAll()
    - Sommario: si procede aggiungendo alla lista principale degli elementi ed alcuni di questi ad una lista creata sul momento per verificare che gli elementi che vengono rimossi invocando removeAll(listasecondaria) sulla lista principale siano solo la prima istanza di quelli presenti. Verifico poi che la lista abbia la dimensione che mi aspetto
    - Test: public boolean removeAll(HCollection c)
    - Design & Description: Il test deve verificare che venga rimossa solamente la prima istanza di ogni elemento contenuta nella HCollection passata come parametro
    - Pre-condizioni: Una nuova HList in cui aggiungo elementi con add() e l'aggiunta di elementi nella lista
    - Post-condizioni: la lista deve tornare ad essere vuota
    - Risultato atteso: La lista vuota o con elementi ad esclusione da quelli che sono stati rimossi dalla chiamata del removeAll tranne le copie, dato che removeAll rimuove solamente la prima istanza degli elementi uguali a quelli presenti nella HCollection. Nel caso in cui la HCollection passata come parametro sia *null* viene lanciata l'eccezione *NullPointerException*.

15. Test metodo list_retainAll()
    - Sommario: si procede aggiungendo alcuni elementi alla lista principale ed alcuni di questi anche ad un'altra lista creata sul momento per verificare che all'invocazione del metodo retainAll(listasecondaria) vengano rimossi solamente gli elementi che sono presenti nella lista secondaria, verifico poi che la grandezza della lista sia coerente con le mie aspettative
    - Test: public boolean retainAll(HCollection c)
    - Design & Description: il test verifica la rimozione di tutte le prime istanze degli elementi che sono presenti nella lista ma non nella HCollection
    - Pre-condizioni: La HCollection creata deve avere elementi validi aggiunti correttamente con add()
    - Post-condizioni: la lista deve tornare ad essere vuota
    - Risultato atteso: il metodo ritornerà true solamente se la dimensione della lista subisce cambiamenti di dimensione dopo l'invocazione del metodo. In quel caso la lista dovrà avere meno oggetti e mantenere quelli che sono presenti anche nella HCollection. Nel caso in cui la HCollection passata come parametro sia *null* viene lanciata l'eccezione *NullPointerException*.

16. Test metodo list_set()
    - Sommario: si procede con l'aggiunta di alcuni elementi nella lista per poi verificare che l'elemento che vado a sovrascrivere con il metodo set sia quello che mi aspettavo dall'ordine in cui gli ho aggiunti
    - Test: public Object set(int index, Object element)
    - Design & Description: il test verifica il rimpiazzo di un elemento contenuto nella lista con un altro elemento passato come parametro della funzione
    - Pre-condizioni: L'indice deve essere contenuto nella lista e l'elemento non deve essere *null*
    - Post-condizioni: la lista deve tornare ad essere vuota
    - Risultato atteso: L'elemento di indice index deve essere rimpiazzato con quello passato come parametro, il metodo ritorna l'oggetto che è stato rimpiazzato. Nel caso in cui l'indice non sia valido o l'oggetto sia *null* viene lanciata l'eccezione *IndexOutOfBoundException* o *NullPointerException*.

17. Test metodo list_size()
    - Sommario: si procede aggiungendo elementi e verificando poi, che la dimensione della lista uguale al numero di elementi che ci ho aggiunto
    - Test: public int size()
    - Design & Description: Il test deve controllare la dimensione della lista in seguito a delle operazioni effettuate su di essa

- Pre-condizioni: il metodo add() deve essere funzionante per poter aggiungere gli elementi nella lista
- Post-condizioni: la lista deve tornare ad essere vuota
- Risultato atteso: Il metodo deve ritornare la dimensione della lista

18. Test metodo list_sublist()
    - Sommario: in questo test verifico innanzitutto che, dopo aver aggiunto degli elementi alla lista, la sublist corrisponda ad un certo range di elementi che mi aspetto di ottenere, verifico poi la corretta creazione di un iteratore per lo scorrimento di tali elementi ottenuti dalla lista. Il resto dei test effettuati per questo metodo in particolare sono gli stessi che sono stati fatti per il resto dei test presenti nella classe ListAdapter. C'è da ricordare che **quando si opera nella sublist i cambiamenti effettuati si riflettono anche sulla lista principale (ma non il contrario)**
    - Test: In questo test viene controllata tutta la sublist
    - Design & Description: Questo test verifica tutta la sublist, dalla sua creazione al toArray, i test avvengono nello stesso modo di quelli fatti per il resto del listAdapter, in questo modo si è potuto risparmiare codice copiando i test e cambiando solamente l'oggetto in cui viene invocato il metodo. L'unico test diverso è quello della creazione della subList (costruttore) in cui si verifica la corretta creazione della struttura dagli elementi presenti nella lista
    - Pre-condizioni: il metodo add() deve essere funzionante per poter aggiungere gli elementi nella lista, questi elementi verranno poi copiati nella sublist
    - Post-condizioni: la lista deve tornare ad essere vuota
    - Risultato atteso: La creazione di una sublist con estremi (from, to) rispettivamente (incluso, escluso) in cui posso effettuare le stesse operazioni e chiamare gli stessi metodi della lista madre, in particolare saranno presenti gli stessi metodi e gli stessi iteratori solamente "adattati" per rispettare i limiti degli indici della sublist. Dopo ogni test case della sublist la sublist stessa viene ripulita tramite il suo metodo clear che la rende così utilizzabile per il test case successivo senza il pericolo di fare operazioni pericolose con gli indici degli elementi ancora presenti nella sublist. Alla fine, quindi avremo una sublist vuota. Nel caso in cui i due indici per la determinazione non siano validi viene lanciata l'eccezione *IndexOutOfBoundsException*.

19. Test metodo list_toArray()
    - Sommario: In questo test si verifica che dopo aver aggiunto alcuni elementi alla lista il metodo toArray della lista ritorni esattamente quelli aggiunti in precedenza, che il metodo toArray di un elemento più grande rispetto a quello della lista si uguale alla lista con aggiunta un elemento nullo e che si riesca a resizare anche riducendo le dimensioni
    - Test: public Object[] toArray(), public Object[] toArray(Object[] a)

- Design & Description: Il test verifica che i metodi copino gli elementi della lista dentro un array creato o passato come parametro al metodo
- Pre-condizioni: il metodo add() deve essere funzionante per poter aggiungere gli elementi nella lista
- Post-condizioni: la lista deve tornare ad essere vuota
- Risultato atteso: un array contenente gli elementi della lista o dell'array passato, nel caso in cui la dimensione dell'array come parametro sia minore di quella della lista, si crea un nuovo array la cui dimensione è come quella della lista per poi copiarci la lista. Nel caso in cui venga passato un array *null* viene lanciata l'eccezione *NullPointerException*

MapTest.java – verranno istanziate 6 coppie chiave valore e un MapAdapter che potremo usare in ogni test case

Di default il MapAdapter viene inizializzato vuoto

Nota: nei test non sono presenti **add** e **addAll** in quanto non sono metodi supportati.

1. Test metodo map_clear()
   - Sommario: si verifica che all'inizio del test la mappa sia vuota, per poi aggiungerci una coppia, verificare l'aggiunta e dopo l'invocazione del metodo clear(), verificare che sia di nuovo vuota
   - Test: public void clear()
   - Design & Description: il test deve controllare che la mappa sia vuota dopo aver chiamato il metodo clear della mappa in seguito all'aggiunta di qualche coppia chiave-valore nella struttura
   - Pre-condizioni: Per l'esecuzione di questo test è richiesto che il metodo put ed isEmpty() funzionino correttamente
   - Post-condizioni: la mappa deve tornare ad essere vuota
   - Risultato atteso: la mappa deve essere priva di qualsiasi elemento al suo interno

2. Test metodo map_containsKey()
   - Sommario: si procede con l'aggiunta di una coppia nella mappa per poi verificare che con la chiamata del metodo la chiave sia contenuta nella mappa stessa.
   - Test: public boolean containsKey(Object key)
   - Design & Description: il test deve controllare che la mappa contenga una determinata chiave dopo aver inserito una coppia chiave-valore
   - Pre-condizioni: Per l'esecuzione di questo test è richiesto che il metodo put ed isEmpty() funzionino correttamente
   - Post-condizioni: la mappa deve tornare ad essere vuota
   - Risultato atteso: il metodo deve ritornare true nel caso in cui nella mappa sia contenuta una chiave uguale a quella passata come parametro, nel caso in cui questa sia *null* viene lanciata l'eccezione *NullPointerException*.

3. Test metodo map_containsValue()
   - Sommario: si procede con l'aggiunta di una coppia nella mappa per poi verificare che con la chiamata del metodo il valore sia contenuto nella mappa stessa.
   - Test: public boolean containsValue()
   - Design & Description: il test deve controllare che la mappa contenga un determinato valore dopo aver inserito una coppia chiave-valore
   - Pre-condizioni: Per l'esecuzione di questo test è richiesto che il metodo put ed isEmpty() funzionino correttamente

- Post-condizioni: la mappa deve tornare ad essere vuota
- Risultato atteso: il metodo deve ritornare true nel caso in cui nella mappa sia contenuto un valore uguale a quello passato come parametro

4. Test metodo map_entrySet()
    - Sommario: si procede con l'aggiunta di alcune coppie nella mappa, per poi creare un Set dalle coppie della mappa e un Iteratore che opera sul Set. Creo poi un oggetto (che chiamo *e*) corrispondente al primo elemento nell'iteratore e verifico che questa coppia sia presente nella mappa, ossia il corretto funzionamento della creazione del Set e dell'Iteratore. Rimpiazzo poi il valore della chiave corrispondente all'oggetto *e* e verifico che questo venga rimpiazzato correttamente anche all'interno della mappa. Faccio poi un controllo sulla chiave dell'elemento successivo nell'iteratore, ossia, dopo aver rimosso una coppia dal Set verifico che questa azione sia stata riflessa anche sulla mappa. Controllo poi con una serie di operazioni che i metodi sul set funzionino correttamente anche passando gli elementi di un ListAdapter come parametro. Verifico infine, dopo aver ripulito e reinserito i valori nella mappa che funzionino correttamente anche i metodi retainAll e toArray.
    - Test: in questo test vengono controllati tutti i metodi presenti nella sottoclasse Entryset
    - Design & Description: Questo test verifica tutta la corretta impostazione della struttura EntrySet, dalla creazione dell' EntrySet, alla corretta esecuzione dei metodi che vengono implementati nella sottoclasse. Il test esegue quindi il corretto funzionamento di tutti i metodi clear, remove, contains, size, isEmpty, toarray, oltre che a verificare anche che l'iteratore contenuto nella sottoclasse operi nel modo corretto tramite i metodi next, hasNext e remove.
    - Pre-condizioni: per l'esecuzione di questo test è necessario che il metodo put sia funzionante nel modo corretto per garantire l'aggiunta delle coppie all'interno della mappa, coppie su cui andrà poi ad operare il Set
    - Post-condizioni: la mappa deve tornare ad essere vuota
    - Risultato atteso: il metodo deve creare un Set in cui posso operare per gestire le coppie al suo interno come nella mappa, in cui le operazioni sul Set generano cambiamenti sulla mappa e viceversa.

5. Test metodo map_equals()
    - Sommario: si procede con l'aggiunta di due coppie nella mappa per poi creare un'altra mappa temporanea e aggiungere le stesse due coppie nella mappa nuova, si verificherà poi che le due mappe risultino uguali.
    - Test: public boolean equals(Object o)
    - Design & Description: il test deve controllare che dati due oggetti questi siano uguali
    - Pre-condizioni: per l'esecuzione di questo test è richiesto che il metodo put ed il metodo putAll funzionino correttamente

- Post-condizioni: la mappa deve tornare ad essere vuota
- Risultato atteso: il metodo deve ritornare true nel caso in cui i due oggetti siano uguali, in ogni altro caso ritornerà false

6. Test metodo map_get()
   - Sommario: si procede aggiungendo una coppia alla mappa e poi con il metodo get verifica di ottenere il valore della coppia aggiunta tramite il passaggio della chiave univoca
   - Test: public Object getKey(Object key)
   - Design & Description: il test deve controllare che invocando il metodo get() passando la chiave venga restituito il valore nella mappa associato
   - Pre-condizioni: per l'esecuzione di questo test è richiesto che il metodo put funzioni correttamente
   - Post-condizioni: la mappa deve tornare ad essere vuota
   - Risultato atteso: il metodo deve ritornare l'oggetto contenente il valore della chiave associata all'invocazione del metodo get(), nel caso in cui la chiave non sia *null* viene lanciata l'eccezione *NullPointerException*

7. Test metodo map_isEmpty()
   - Sommario: in questo test si verifica che la dimensione della mappa sia 0 nel caso in cui non sia stata riempita con elementi, nel nostro caso aggiungiamo una coppia e poi la rimuoviamo per poi verificare che la dimensione sia tornata 0.
   - Test: public boolean isEmpty()
   - Design & Description: il test deve verificare che all'inizio la mappa sia vuota
   - Pre-condizioni: la mappa deve essere inizializzata vuota
   - Post-condizioni: la mappa deve rimanere vuota
   - Risultato atteso: il metodo deve ritornare true nel caso in cui la mappa sia effettivamente vuota, false altrimenti

8. Test metodo map_keySet()
   - Sommario: si procede con l'aggiunta di alcune coppie nella mappa, per poi creare un Set **delle chiavi** dalle coppie della mappa e un Iteratore che opera sul Set. Creo poi un oggetto (che chiamo *e*) corrispondente al primo elemento nell'iteratore e verifico che questa coppia sia presente nella mappa, ossia il corretto funzionamento della creazione del Set e dell'Iteratore. Rimpiazzo poi il valore della chiave corrispondente all'oggetto *e* e verifico che questo venga rimpiazzato correttamente anche all'interno della mappa. Faccio poi un controllo sulla chiave dell'elemento successivo nell'iteratore, ossia, dopo aver rimosso una coppia dal Set verifico che questa azione sia stata riflessa anche sulla mappa. Controllo poi con una serie di operazioni che i metodi sul set funzionino correttamente anche passando gli elementi di un ListAdapter come

parametro. Verifico infine, dopo aver ripulito e reinserito i valori nella mappa che funzionino correttamente anche i metodi retainAll e toArray.
- Test: in questo test vengono controllati tutti i metodi presenti nella sottoclasse KeySet
- Design & Description: Questo test verifica tutta la corretta impostazione della struttura KeySet, dalla creazione del KeySet, alla corretta esecuzione dei metodi che vengono implementati nella sottoclasse. Il test esegue quindi il corretto funzionamento di tutti i metodi clear, remove, contains, size, isEmpty, toarray, oltre che a verificare anche che l'iteratore contenuto nella sottoclasse operi nel modo corretto tramite i metodi next, hasNext e remove.
- Pre-condizioni: per l'esecuzione di questo test è necessario che il metodo put sia funzionante nel modo corretto per garantire l'aggiunta delle coppie all'interno della mappa, coppie su cui andrà poi ad operare il KeySet
- Post-condizioni: la mappa deve tornare ad essere vuota
- Risultato atteso: il metodo deve creare un KeySet in cui posso operare per gestire le chiavi al suo interno come nella mappa, in cui le operazioni sul KeySet generano cambiamenti sulla mappa e viceversa.

9. Test metodo map_put()
    - Sommario: in questo test si aggiunge con il metodo put(key,value) una coppia nella mappa, per poi verificare che la lunghezza di questa sia aumentata di 1. Si verifica poi che nel caso di un inserimento di una coppia con chiave uguale la coppia venga sovrascritta o che esistano correttamente due coppie con chiave diversa ma valore uguale
    - Test: public Object put(Object key, Object value)
    - Design & Description: il test deve verificare il corretto funzionamento della funzione put(), in particolare che l'oggetto venga correttamente aggiunto alla mappa e che la dimensione di quest'ultima aumenti, o che sostituisca una coppia chiave-valore già inserita precedentemente (nel caso di chiave uguale)
    - Pre-condizioni: per l'esecuzione di questo test è richiesto che il metodo size e get funzionino correttamente
    - Post-condizioni: la mappa deve tornare ad essere vuota
    - Risultato atteso: il metodo deve ritornare il valore della entry chiave-valore che è stata sostituita con il nuovo inserimento o *null* nel caso non sia stato sostituito alcun elemento all'interno della mappa

10. Test metodo map_putAll()
    - Sommario: si procede con la creazione di una nuova mappa e con l'inserimento nella mappa di alcune coppie chiave-valore, per poi inserire tutte le coppie

nella mappa principale (inizializzata vuota) tramite il metodo putAll(), si verifica poi il corretto inserimento delle coppie tramite i metodi get(key) invocato su tutte le chiavi inserite nella mappa.
- Test: public void putAll(HMap t)
- Design & Description: il test deve verificare che, costruita una mappa ad aggiunti degli elementi a questa, gli elementi della mappa creata vengano aggiunti tutti alla mappa principale su cui viene invocato il metodo putAll
- Pre-condizioni: per l'esecuzione di questo test è richiesta la creazione di una mappa a cui vengono aggiunti degli elementi (metodo put funzionante)
- Post-condizioni: la mappa deve tornare ad essere vuota
- Risultato atteso: il metodo deve aggiungere correttamente tutte le entry contenute nella mappa passata nella mappa principale, richiamando il metodo put() incrementando la dimensione o sostituendo la entry nel caso di chiavi uguali. Nel caso in cui la mappa passata come parametro sia *null* viene lanciata l'eccezione *NullPointerException*

11. Test metodo map_remove()
    - Sommario: si procede con l'aggiunta di una coppia nella mappa, per poi assicurarsi che all'invocazione del metodo remove(key) venga rimosso il valore associato alla chiave passata come parametro. Si prova poi a rimuovere nuovamente la coppia inserita all'inizio, ma questa volta, giustamente, il metodo lancia eccezione perché la mappa non contiene la chiave da rimuovere.
    - Test: public Object remove(Object key)
    - Design & Description: il test deve verificare che, dopo aver aggiunto una entry nella mappa, questa venga rimossa correttamente invocando il metodo remove() e passando come parametro la chiave della entry da rimuovere
    - Pre-condizioni: per l'esecuzione di questo test è necessario che il metodo put sia correttamente funzionante
    - Post-condizioni: la mappa deve tornare ad essere vuota
    - Risultato atteso: il metodo deve ritornare la entry che è stata rimossa, nel caso in cui la chiave della entry da rimuovere non sia presente nella mappa o sia *null*, viene lanciata l'eccezione *NullPointerException*

12. Test metodo map_size()
    - Sommario: in questo test si procede con l'aggiunta di alcune coppie per poi verificare che la mappa (inizializzata vuota) abbia una dimensione pari al numero di elementi aggiunti in precedenza.
    - Test: public int size()
    - Design & Description: il test deve verificare che, dopo aver aggiunto(o tolto) degli elementi alla mappa, quest'ultima veda incrementata(o decrementata) la sua dimensione
    - Pre-condizioni: per l'esecuzione di questo test è necessario che il metodo put sia correttamente funzionante

- Post-condizioni: la mappa deve tornare ad essere vuota
- Risultato atteso: il metodo deve ritornare correttamente la dimensione della mappa

13. Test metodo map_values()
    - Sommario: si procede con l'aggiunta di alcune coppie nella mappa, per poi creare una Collection **dei valori** dalle coppie della mappa e un Iteratore che opera sulla Collection. Creo poi un oggetto (che chiamo *e*) corrispondente al primo elemento nell'iteratore e verifico che questa coppia sia presente nella mappa, ossia il corretto funzionamento della creazione della Collection e dell'Iteratore. Rimpiazzo poi il valore della chiave corrispondente all'oggetto *e* e verifico che questo venga rimpiazzato correttamente anche all'interno della mappa. Faccio poi un controllo sul valore dell'elemento successivo nell'iteratore, ossia, dopo aver rimosso una coppia dalla Collection verifico che questa azione sia stata riflessa anche sulla mappa. Controllo poi con una serie di operazioni che i metodi sul set funzionino correttamente anche passando gli elementi di un ListAdapter come parametro. Verifico infine, dopo aver ripulito e reinserito i valori nella mappa che funzionino correttamente anche i metodi retainAll e toArray.
    - Test: in questo test vengono controllati tutti i metodi presenti nella sottoclasse ValueCollection
    - Design & Description: Questo test verifica tutta la corretta impostazione della struttura ValueCollection, dalla creazione del ValueCollection, alla corretta esecuzione dei metodi che vengono implementati nella sottoclasse. Il test esegue quindi il corretto funzionamento di tutti i metodi clear, remove, contains, size, isEmpty, toarray, oltre che a verificare anche che l'iteratore contenuto nella sottoclasse operi nel modo corretto tramite i metodi next, hasNext e remove.
    - Pre-condizioni: per l'esecuzione di questo test è necessario che il metodo put sia funzionante nel modo corretto per garantire l'aggiunta delle coppie all'interno della mappa, coppie su cui andrà poi ad operare il ValueCollection
    - Post-condizioni: la mappa deve tornare ad essere vuota
    - Risultato atteso: il metodo deve creare e ritornare una Collection in cui posso operare per gestire le chiavi al suo interno come nella mappa, in cui le operazioni sulla Collection generano cambiamenti sulla mappa e viceversa.