10. Laboratorio 10

Ciascuno di questi esercizi richiede l'implementazione di diversi metodi. Quindi un esercizio puo' richiedere anche un'ora o due di tempo, soprattutto le prime volte che si implementano queste strutture.

Esercizio 1

Realizzare una classe SortedArraySortedMap che implementi l'interfaccia SortedMap utilizzando un array riempito in parte in cui le associazioni sono mantenute ordinate per chiave. Riflettere sulla complessita' dei metodi di inserimento/rimozione/ricerca. Poiche' SortedMap estende Map che a sua volta estende Container, tutte e tre le interfacce dovranno essere presenti nella cartella in cui compilate e tutti i metodi in esse definite devono essere implementati. Potete testare il funzionamente con SortedMapTester, oppure creare voi stessi un tester.

Esercizio 2

Realizzare una classe ArrayMultiMap che implementi l'interfaccia MultiMap con un array riempito in parte in cui l'ordine degli elementi non e' importante. Riflettere sulla complessita' dei metodi di inserimento/rimozione/ricerca. Potete provare la classe con il tester MultiMapTester

Esercizio 3

Realizzare una classe SortedArrayMultiMap che implementi l'interfaccia MultiMap con un array riempito in parte in cui le associazioni sono mantenute in ordine di chiave.
Riflettere sulla complessita' dei metodi di inserimento/rimozione/ricerca e le differenze con l'implementazione dell'esercizio 2. Potete provare la classe con il tester SortedMultiMapTester.

Esercizio 4

Realizzare una classe ArrayTable100 che implementi l'interfaccia Table fissando la dimensione della tabella a 100. Nel caso in cui si provi ad utilizzare chiavi con valore <0 oppure >=100 lanciare l'eccezione InvalidPositionTableException (da realizzare derivandola da RuntimeException). Potete provare la classe con il tester TableTester.java o scrivere un vostro tester.

Esercizio 5

Implementare l'**ADT Deque** la cui interfaccia e' descritta qui con una classe **ArrayDeque**.

Utilizzare un array gestito come coda circolare in modo che ogni operazione sia O(1), cioè tempo-costante. Fare attenzione alle condizioni di riempimento/svuotamento (suggerimento: e' possibile utilizzare una variabile di esemplare **size** che tenga traccia del numero di elementi presenti).

Attenzione! Last funziona come abbiamo visto per la coda, mentre first indichera' la prima posizione in cui inserire in testa agli elementi gia' presenti. Quando raggiunge l'estremo sinistro (cella di indice 0) deve proseguire dall'ultima (cella di indice v.length-1) per garantire la circolarita' anche in quella direzione.

Il ridimensionamento puo' essere fatto come abbiamo visto per la coda circolare, tuttavia la cosa importante e' che venga mantenuto l'ordine relativo tra gli elementi,
non gli indici effettivamente occupati da tali elementi. Per questo puo' essere conveniente, quando si devono copiare gli elementi nel nuovo array, partire dalla sua prima posizione, cosi' che gli elementi, in ordine occupino le posizioni da 0 a size-1 nel
nuovo array. In questo modo e' piu' facile stabilire i valori da associare a
first (newV.length-1) e last (size).

Sovrascrivere nella classe ArrayDeque il metodo **toString()** in modo che restituisca una stringa in "formato array" dell'intero array. Ad esempio se la dimensione fisica e' 6 e nelle prime tre celle ci sono i numeri 1, 2 e 3 la stringa restituita sara': "[1 2 3 null null null]".

E' possibile testarne il funzionamento con il bytecode ProfDequeTester.class fornito.

Scrivere una classe **DequeTester** di prova che:

- crei tre esemplari della classe ArrayDeque di nome uno, due e tre
- legga dall'ingresso standard una sequenza di numeri interi (uno per riga) e li inserisca alla fine della coda uno (con addLast)
- svuoti la coda uno dalla fine (con removeLast) trasferendone il contenuto all'inizio della coda due (con AddFirst)
- svuoti la coda due dall'inizio (con removeFirst) inserendo i dati alla fine della coda tre

- svuoti infine la coda tre dall'inizio scrivendo i dati sull'uscita standard, uno per riga.

I dati in uscita dovranno essere nello stesso ordine dei dati in ingresso.

Esercizio 6

Implementare una pila DequeStack utilizzando come struttura di memorizzazione ArrayDeque.

Esercizio 7

Implementare una "Pila sicura" in cui gli elementi inseriti possono solo essere stringhe.

Se si prova a inserire un elemento che non e' una stringa, lanciare IllegalArgumentException.

Esercizio 8

Obiettivo: imparare a gestire le operazioni di una lista concatenata.

Data la struttura dati lista concatenata implementata nella classe LinkedList.java (che fornisce i metodi: addFirst, addLast, removeFirst, removeLast, toString, getIterator) implementare una classe eseguibile LinkedListTester.java in cui:

1) Si crei un esemplare di LinkedList inizialmente vuota

2) Si chieda all'utente di inserire un numero intero n

3) Si popoli la lista concatenata generando n numeri casuali con valori da 1 a 100. In particolare il primo numero andra' inserito all'inizio della lista, il secondo alla fine, il terzo all'inizio e cosi' via... Per ogni inserimento stampare a video il valore da inserire e il contenuto della lista concatenata dopo ogni inserimento (sfruttare il fatto che LinkedList implementi toString) e verificare che l'inserimento avvenga correttamente.

4) Chiedere all'utente dopo quale elemento presente nella lista si vuole inserire il valore 1000. Creare un'istanza di ListIterator sull'istanza di lista concatenata su cui si sta lavorando (sfruttare il metodo getIterator()) e scansionare la lista (in modo opportuno con i metodi hasNext() e next()). Quando si individua l'elemento inserito dall'utente, invocare il metodo di inserimento dell'iteratore (add(Object o)) passando il valore 1000. Alla fine della scansione stampare il contenuto della lista concatenata e verificare che 1000 sia stato inserito nella giusta posizione.

5) Chiedere all'utente un elemento da eliminare. Creare una nuova istanza dell'iteratore, scansionare la lista e, quando si individua l'elemento indicato, eliminarlo dalla lista con il metodo dell'iteratore (remove()). Alla fine della scansione stampare il contenuto della lista concatenata e verificare che l'elemento sia stato eliminato.

6) Rimuovere il primo elemento della lista e stamparne il contenuto per verificare che l'operazione sia andata a buon fine.

7) Rimuovere l'ultimo elemento della lista e stamparne il contenuto per verificare che l'operazione sia andata a buon fine.

Es: Se i numeri generati casualmente, in ordine, sono: 45,2,67,34,77,21 la lista avra' questo contenuto: 77 67 45 2 34 21

Se l'utente inserisce il numero 45, il valore 1000 verra' introdotto dopo di esso:

77 67 45 1000 2 34 21

Se l'untente inserisce il numero 2, questo verra' rimosso

77 67 45 1000 34 21

Eliminazione del primo elemento

67 45 1000 34 21

Eliminazione dell'ultimo elemento

67 45 1000 34

Esercizio 9

Scrivere una classe eseguibile ListReverse che inizializzi una LinkedList con i numeri da 0 a n-1, dove n e' un parametro letto da input. Stampare il contenuto della lista in ordine inverso con un metodo statico ricorsivo. Si ricorda che per scandire una lista concatenata e' necessario utilizzare un iteratore, quindi la firma del metodo statico sara':

public static void printReverse(ListIterator l)