Progetto LinkedList

Punto 1

Utilizzando esclusivamente i metodi del ListIterator<T> si implementa una variante del metodo static sort previsto nell'interfaccia List<T>: il metodo in questione basa il suo funzionamento sull'algoritmo di ordinamento "insertion sort".

Il metodo riceve in input una lista generica in <T>, un oggetto comparatore c ed una flag booleana.

Il metodo applica, alla lista passata come parametro, l'ordinamento secondo bubbleSort se la flag risulta essere false, altrimenti applica l'ordinamento secondo insertionSort: gli elementi costituenti la lista verranno confrontati ed ordinati secondo l'oggetto comparatore passato come parametro.

L'idea di base, sulla quale si fonda la costruzione del metodo, poggia sulla classica implementazione del metodo insertionSort, adattato ovviamente al caso della Linked List.

Se la lista risulta essere vuota o costituita da un solo elemento, si ha un caso particolare e si restituisce la lista stessa (che è già ordinata).

Altrimenti, ad ogni iterazione di un ciclo, che termina al finire degli elementi della lista, si prende come riferimento un elemento x da ordinare, a partire dalla posizione i = 1, sino ad arrivare a i = size(si noti che alla posizione 0 l'elemento non segue alcun valore).

Al fine di scandire gli elementi della lista, e spostarsi opportunamente su di essa, prelevandone i valori, si utilizza un list iterator dichiarato di volta in volta a partire dalla posizione i, è opportuno infatti notare che gli elementi della sottolista che vanno da 0 a i-1 saranno ordinati.

Al'interno di un ulteriore ciclo, annidato nel precedente, l'elemento x viene confrontato, a partire da destra verso sinistra, con i suoi precedenti, memorizzati, di volta in volta, all'interno di una variabile y: verranno spostati a destra, di una posizione, gli elementi maggiori del valore x preso come riferimento.

Il ciclo interno prosegue, quindi, fin quando il valore y, che di volta in volta viene memorizzato attraverso l'iteratore, risulta maggiore del valore x (elemento da ordinare nella iterazione corrente), altrimenti, se l'elemento precedente ad x non risulta essere maggiore, non viene effettuato alcuno scambio e si passa direttamente all'elemento successivo terminando l'iterazione nel ciclo innestato, in quanto essendo la sottolista ordinata da 0 a i-1, anche i valori a ritroso risulteranno non maggiori.

A termine dell'iterazione del ciclo innestato, ogni volta, il valore di x, verrà inserito, all'interno della lista, nella posizione corrente dell'iteratore, si avrà di conseguenza una sottolista ordinata da sinistra verso destra sino al valore corrente da ordinare.

Al termine del ciclo più esterno, l'intera lista risulterà ordinata.

Si propone all'interno del main della classe concreta LinkedList, un test del metodo sopra trattato, effettuato sulla linkedList "test1".

Punto 2

Utilizzando esclusivamente i puntatori si implentano i metodi proposti in veste iterativa, all'interno della classe concreta LinkedList.

I metodi in questione risultato essere addFirst, addLast, getFirst, getLast, removeFirst, removeLast.

- Il metodo addFirst aggiunge nella prima posizione della lista l'elemento *"e"* passato come parametro: viene quindi creato un nuovo nodo *"n"* che ha: come valore l'elemento stesso (n.info = e) , come successivo il primo nodo della lista in questione (n.next = first), ovvero l'attuale testa, e come precedente il valore nullo (n.prior = null).
  Nel caso in cui la lista non fosse vuota (first != null) si pone uguale ad *n* il precedente della testa attuale della lista(first.prior = n); nel caso in cui la lista fosse vuota , l'elemento inserito, sarà anche l'ultimo elemento della lista e di conseguenza si pone la coda della lista uguale proprio all'elemento inserito(last =n).
  A prescidendere dalla lista il nodo appena inserito costituirà la nuova testa della lista (first = n).
  Si incrementano inoltre la lunghezza della lista (size) e la variabile, modCounter, relativa al numero di modifiche effettuate sulla lista(utilizzata al fine di gestire la ConcurrentModificationException dell'iteratore).

- Il metodo addLast aggiunge nell'ultima posizione della lista l'elemento "e" passato come parametro: viene quindi creato un nuovo nodo *"n"* che ha: come valore l'elemento stesso (n.info = e), come successivo il valore nullo (n.next = null) e come precedente l'ultimo nodo della lista in questione (n.prior = last), ovvero l'attuale coda.
  Nel caso in cui la lista non fosse vuota (first != null) si pone uguale ad *n* il successivo della coda attuale della lista(last.next = n); nel caso in cui la lista fosse vuota , l'elemento inserito, sarà anche primo elemento della lista e di conseguenza si pone la testa della lista uguale proprio all'elemento inserito(first =n).
  A prescidendere dalla lista il nodo appena inserito costituirà la nuova coda della lista (last = n).
  Si incrementano inoltre la lunghezza della lista (size) e la variabile, modCounter, relativa al numero di modifiche effettuate sulla lista(utilizzata al fine di gestire la ConcurrentModificationException dell'iteratore).

- Il metodo getFirst restituisce il primo elemento appartenente alla lista in questione. Se quest'ultima risulta essere vuota (first == null), si restituisce il valore nullo, altrimenti, si restituisce il valore relativo alla testa della lista (first.info). Nessuna modifica viene effettuata sulla lista.

- Il metodo getLast restituisce l'ultimo elemento appartenente alla lista in questione. Se quest'ultima risulta essere vuota (first == null), si restituisce il valore nullo, altrimenti, si restituisce il valore relativo alla coda della lista (last.info). Nessuna modifica viene effettuata sulla lista.

- Il metodo removeFirst rimuove il primo elemento appartenente alla lista e lo restituisce. Se quest'ultima risulta essere vuota, non viene effettuata alcuna azione ed il metodo termina, altrimenti, si memorizza l'informazione associata alla testa della lista all'interno di una variabile *"cor"*, si pone l'attuale testa della lista uguale al nodo successivo alla testa stessa (first = fist.next), si pone il precedente della testa pari al valore nullo (first.prior = null) e si restituisce la variabile *cor*.
  Si decrementano inoltre la lunghezza della lista (size) e la variabile modCounter.

- Il metodo removeLast rimuove l'ultimo elemento appartenente alla lista e lo restituisce. Se quest'ultima risulta essere vuota, non viene effettuata alcuna azione ed il metodo termina, altrimenti, si memorizza l'informazione associata alla coda della lista all'interno di una variabile *"cor"*, si pone l'attuale coda della lista uguale al nodo precedente alla coda stessa (last = last.prior), si pone il successivo della coda pari al valore nullo (last.next = null) e si restituisce la variabile *cor*. Si decrementano inoltre la lunghezza della lista (size) e la variabile modCounter.

Si propone all'interno del main della classe concreta LinkedList, un test dei metodi sopra trattati, effettuato sulla linkedList "test2".

Punto 3

Si implementano, all'interno della classe concreta LinkedList, i metodi nextIndex() e previousIndex(), i quali restituiscono rispettivamente l'indice dell'elemento che verrebbe restituito da una chiamata a next() o a previous().

L'idea di base sulla quale si fonda la costruzione di questi due metodi, in questa implementazione, si fonda sul conoscere in ogni momento la posizione relativa all'elemento corrente della lista, attraverso questa posizione, diviene possibile conoscere l'indice dell'elemento prossimo e successivo.

A tal proposito si definisce una variabile di classe *"posCor"* che di fatto punterà all'elemento che verrebbe restituito nel caso in cui si eseguisse una previous.

La variabile posCor verrà inizializzata a -1 nel caso in cui l'iteratore venga creato a partire dall'inizio della lista; verrà invece inizializzata alla posizione *pos-1*, nel caso in cui l'iteratore venga creato a partire da una specifica posizione *pos*.

Il metodo nextIndex restituirà il valore posCor+1 che corrisponderà all'elemento che verrebbe restituito a seguito di una next(), si noti che nel caso in cui si fosse in ultima posizione, pos+1 corrisponderebbe alla size della lista;

Il metodo previousIndex restituirà -1 nel caso in cui si fosse nella prima posizione della lista (posCor<0); altrimenti restituità posCor che corrisponderà all'elemento che verrebbe restituito a seguito di una previous().

Nel momento in cui si esegue una next, la variabile posCor viene incrementata di uno, altrimenti, se viene effettuata una previous, posCor viene decremenetata di uno.

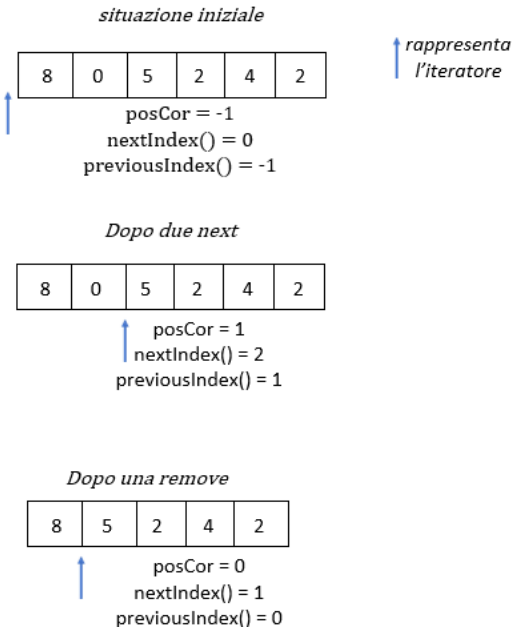
Estremamente importante, è che i due metodi dovranno adattarsi ad eventuali remove eseguito attraverso l'iteratore, ed inoltre, è necessario determinare se l'operazione di rimozione avviene a seguito di una next o di una previous: si ricorda che la remove eseguita attraverso l'iteratore, rimuove l'ultimo elemento restituito da una next o da una previous.

Si distinguono a tal proposito due casi:

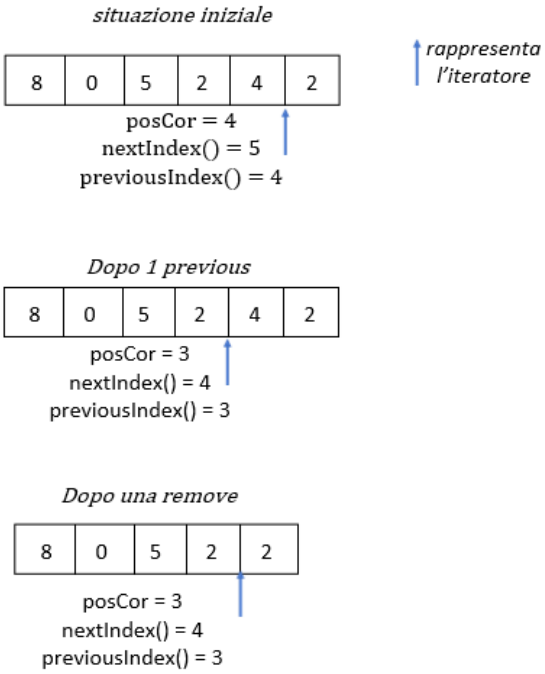
Remove preceduta da next: la next incrementa posCor di 1, la remove elimina l'elemento appena restituito dalla next, è necessario quindi decrementare posCor di 1.

If(lastMove == Move.FORWARD)
posCor--;

Si utilizza l'enumerazione già disponibile che tiene traccia della direzione dell'ultimo spostamento.



Remove preceduta da previous: la previous decrementa posCor di 1, la remove elimina l'elemento appena restituito dalla previous, alla destra di posCor, è necessario quindi lasciare inalterato posCor



Si propone all'interno del main della classe concreta LinkedList, un test dei due metodi sopra trattati, effettuato sulla linkedList "test3".