Progetto 2: LinkedList

Interfaccia List

I metodi default sono concretizzati con l'ausilio dell'iteratore non ancora concreto, ma comunque utilizzabile nell'interfaccia, dato che verrà poi completato nella classe LinkedList.
Grazie all'iteratore si può già scandire la lista e, ad esempio, contare gli elementi tramite un contatore (per il metodo size()) o fare la ricerca lineare di un elemento (per remove())

```
default int size(){
  int size = 0;
  for(T t : this) size++;
  return size;
}//size
```

```
default void remove( T e ){
  Iterator<T> it = this.iterator();
  while(it.hasNext()){
    T t = it.next();
    if(t.equals(e)) it.remove(); //
  }
}//remove
```

I metodi add(T e),addFirst(T e),addLast(T e),sort(Comparator<T> c) e, appunto, listIterator() o iterator non possono essere concretizzati nell'interfaccia come metodi default perché dipendono dalla specifica struttura dati sottostante.

Classe astratta AbstractList

Nella classe astratta faccio l'override di tre metodi della classe Object, ovvero equals(Object o), hashCode() e toString(). Anche per equals e toString sfrutto l'iteratore in modo da poter scorrere la lista.

Classe concreta LinkedList

Nella classe concreta innanzitutto è importante menzionare le variabili d'istanza: *inizio*, *fine* e *size*. Nodo<E>, dove E sarà dello stesso tipo parametrizzato di T, contiene i puntatori agli elementi contigui (*next* e *previous*) e il contenuto di ogni elemento della LinkedList(info), ovvero ogni nodo. Quasi tutti i metodi propri della LinkedList rimandano a un metodo dell'Iteratore. L'unico metodo un po' più elaborato è il metodo di ordinamento, basato su bubble-sort.
Il metodo si basa su un ciclo esterno che scorre finché *scambio* è true, ovvero finché avvengono degli scambi fra due nodi contigui. L' int *limite* indica l'indice fino al quale possono avvenire scambi, infatti inizialmente *limite* corrisponde a *size.* Dentro il ciclo esterno vengono istanziati due iteratori, uno a partire da -1 e l'altro da 0, e parte un altro ciclo interno che fa scorrere parallelamente i due iteratori, confrontando a ogni passo due nodi contigui. Lo swap avviene se il primo elemento é maggiore del secondo, ovvero del suo successore. Il ciclo si ferma quando é finita la lista o quando il secondo iteratore é arrivato al limite. Limite viene decrementato ogni volta che viene raggiunto perché il massimo ha raggiunto la sua posizione definitiva.

```
while(li1.hasNext() && li2.hasNext() && li2.nextIndex() < limite) {
    next1 = li1.next(); next2 = li2.next();
    if(c.compare(next1, next2) > 0) { //se il primo elemento é maggi
        //SWAP
        li1.set(next2); li2.set(next1);
        if(li2.nextIndex() == limite) limite--;
        scambio = true;
```

Inner-Class Iteratore

Ha come variabili d'istanza 3 nodi: *nexti*, *previousi* e *processato*, un oggetto di tipo enumerativo Direzione e una variabile *cursore* di tipo int. *Nexti* e *previousi* hanno questi nomi per distinguerli da *next* e *previous* e indicano gli elementi contigui al puntatore dell'iteratore, che idealmente sta sempre fra due elementi.
La variabile *processato* indica l'elemento appena tornato da una next() o una *previous().*

Il costruttore dell'iteratore pone idealmente il cursore davanti al primo elemento, per cui ho deciso di fare partire cursore da -1 e *nexti* sarà la testa della lista, ovvero *inizio*.
La next() assegna al precedente il successivo, a cui a sua volta assegna quello dopo ancora.

```
Nodo temp = previousi;
try{
    previousi = nexti; nexti = nexti.next;
}catch(NullPointerException e) {
    previousi = temp;
    return null;
}
```

La previous() assegna al successivo il precedente, a cui a sua volta assegna quello prima ancora

```
Nodo temp = nexti;
try {
    nexti = previousi; previousi = previousi.previous;
}catch(NullPointerException e) {
    nexti = temp;
    return null;
}
```

Ovviamente se viene sollevata un'eccezione, ad esempio nel caso in cui si é arrivati alla fine o all'inizio della lista, nulla viene cambiato e viene restituito null.
Alla fine del metodo viene aggiornato *cursore* e viene assegnato a *processato* l'elemento appena sorpassato, ovvero *previousi* per la next() e *nexti* per la previous(), ed esso viene poi restituito.

hasNext() e hasPrevious() verificano che *nexti* e *previousi* non siano null.
nextIndex() e previousIndex() ritornano il cursore rispettivamente incrementato e decrementato.

Una add(T e) inserisce un elemento fra il cursore e il nodo precedente.
La add(T e) dell'iteratore presenta 4 diverse situazioni:

1. Inserimento in lista vuota
2. Inserimento in testa
3. Inserimento in coda
4. Inserimento fra due nodi

Il **primo** caso si verifica quando *previousi* e *nexti* sono entrambi null, perciò è la condizione del primo if. *Inizio* e *fine* prendono il contenuto dell'elemento da inserire e il cursore va a 0, dato che deve sempre trovarsi dopo l'elemento appena inserito.
Il **secondo** caso si verifica quando solo il nodo precedente è null. Il nuovo elemento sarà il nuovo *previousi*, ovvero l'elemento restituito dalla prossima previous() e nexti, che in questo caso equivale ad inizio, sarà il *next* del nuovo elemento. Ma oltre a diventare il nuovo *previousi*, il nuovo elemento diventa anche la nuova testa della lista. La seconda istruzione in foto in realtà è superflua, ma è una questione di ordine mentale.

```
nuovo.next = nexti; nuovo.previous = previousi;
previousi = nuovo;
inizio = nexti.previous = previousi;
```

Il **terzo** caso si verifica quando solo il nodo successivo é null. I passaggi sono analoghi al caso precedente. Il nuovo elemento avrà come *previous* la coda della lista e diventerà a sua volta la nuova coda.

```
nuovo.previous = fine; nuovo.next = nexti;
fine = previousi.next = nuovo; //Faccio te
previousi = fine;
```

Il **quarto** e ultimo caso riguarda l'inserimento fra due nodi. I passaggi sono analoghi ai casi precedenti, ma con la differenza che nuovo non può andare a finire su uno degli estremi della lista, perciò diventa il successore di previousi e il predecessore di nexti.

```
nuovo.next = nexti; nuovo.previous = previousi;
previousi.next = nexti.previous = nuovo;
previousi = nuovo;
```

Ciò che differenzia i quattro casi è l'aggiornamento del cursore. Nei primi due cursore viene posto a 0, nel terzo a size – 1 e nel quarto incrementato.

La set(T e) modifica il valore corrente di *processato,* che è diverso da null solo dopo una next() o una previous(). Non si può, infatti, eseguire una set(T e) dopo una add(T e) o una remove() o una stessa set(T e), come non si possono eseguire due remove() di seguito o una remove() dopo una add(T e). Tant'è che, alla fine di add(T e), remove() e set(), *processato* viene posto a null, e all'inizio di remove() e set(T e) viene controllato che *processato* non sia null, perché se lo è si solleva una IllegalStateException.

La remove() si divide in tre casi in basi allo stato della variabile *direzione*, che cambia in base al fatto se l'ultimo movimento é avvenuto con una next() o una previous(). Se direzione è FORWARD, allora, devo fare il bypass di puntatori sul nodo precedente, ovvero *previousi*, giacché è esso l'elemento appena processato. Quindi il *previous* del prossimo nodo diventa quello prima di *previousi* e il predecessore dell'elemento da eliminare avrà come successore non più *previousi*, ma quello dopo, facendo saltare così tutti i collegamenti a *previousi*.

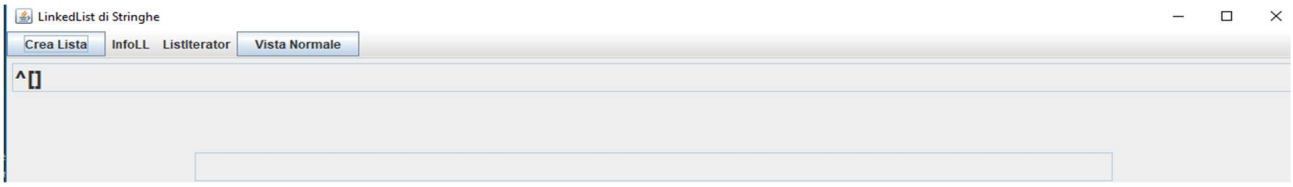
```
size--; cursore--;
if(nexti != null) nexti.previous = previousi.previous;
if(previousi != null) {
    if(previousi.previous == null) inizio = nexti;
    else previousi.previous.next = nexti;
}
previousi = previousi.previous;
```

Se direzione é BACKWARD, i passaggi sono speculari, bisogna bypassare il nodo successivo, ovvero *nexti*. Se direzione é UNKNOWN, metto a null processato. In tutti e tre i casi *size* e *cursore* vengono decrementati.

Interfaccia Grafica

Per quanto riguarda l'interfaccia, essa si presenta in due modalità:
Modalità normale

Modalità iteratore



Queste due modalità sussistono perché, una volta acceso un iteratore, aggiungere elementi con una add() della lista può sollevare una ConcurrentModificationException. Perciò, una volta acceso un iteratore, é possibile usare solo metodi dell'iteratore o chiedere informazioni sulla lista, o pulire la lista, ma non modificarla esternamente all'iteratore stesso.
Nei menu c'è un bottone per ogni metodo.
Gli elementi del JFrame sono la JMenuBar *menubar* con i relativi menu, il JPanel *testo1* che contiene il JTextField *lista*, spazio occupato appunto dalla LinkedList, il JPanel *testo2* che contiene il JTextField *stato*, casella occupata dall'ultimo elemento restituito o, eventualmente, da un'eccezione.

Ho deciso di organizzare gli ascoltatori in tipi di operazioni e assegnare a ogni ascoltare diversi bottoni, fra cui a ogni chiamata dell'actionPerformed(ActionEvent e) si può scegliere attraverso una cascata di if-else, verificando con il metodo getSource():

1. **AscoltatoreMod** – Ascoltatore per i bottoni che corrispondono a metodi della LinkedList che modificano la lista, senza ricevere un elemento in input. Alla fine del metodo ne viene stampata la nuova versione.
2. **AscoltatoreElem** – Ascoltatore per i metodi che prendono un elemento in input come argomento. Uso un JOptionPane (per far immettere l'elemento all'utente) racchiuso in un ciclo for(;;). Una volta presa la stringa controllo lo stato del booleano *tipo*, inizializzato come variabile d'istanza, che é true se all'inizio l'utente ha scelto di costruire una lista di stringhe, false se di interi. *Tipo* mi permette di decidere se entrare nel blocco try-catch nel caso sia false per controllare il valueOf(String a) e in caso chiedere di reinserire l'intero; se invece *tipo* é true, controllo che la stringa non abbia virgole o spazi all'interno. Questo verrà spiegato in seguito. Se invece, in un caso o nell'altro, la verifica va a buon fine, interrompo il ciclo e passo l'elemento al metodo del bottone che é source di quell'evento.
3. **AscoltatoreInfo** – Ascoltatore per i metodi che semplicemente chiedono informazioni alla lista. Alla fine dell'actionPerformed di questo ascoltatore, infatti, la lista non verrà ristampata, ma viene mostrato all'utente l'informazione richiesta tramite un messageDialog di JOptionPane.
4. **AscoltatoreLit** – Ascoltatore per i metodi del listIterator. A ogni metodo richiamato, aggiusto la visibilità dei bottoni. Attivata una next(), ad esempio, diventa visibile il bottone per la remove() e la set(), che magari erano stati resi invisibili da una remove() chiamata subito prima. Alla fine viene messo in *stato* l'elemento processato e ristampata la lista.
5. **AscoltatoreVistaI** – Ascoltatore per attivare la modalità iteratore. Se premuto, spariranno tutti i bottoni relativi a metodi che potrebbero sollevare una ConcurrentModificationException e viene inizializzato un iteratore, facendo scegliere all'utente se da una posizione arbitraria (presa da input) o di default.
6. **AscoltatoreVistaN** – È speculare a quello precedente. Viene ripristinata la visibilità dei bottoni dei metodi che modificano la lista e viene nascosto il menu con i metodi relativi all'iteratore.

È importante, infine, il metodo toStringI(String l), che serve a stampare la lista con il cursore al posto giusto anche graficamente. In pratica sfrutto uno StringBuilder *ris* per poter modificare la stringa e uno StringTokenizer *st* per scandirla fino al punto in cui va modificata.
st ha come separatori virgola e spazio, per questo all'interno delle stringhe da inserire nella lista non ci possono essere. A ogni passo *indice* si trova sul primo carattere del prossimo elemento nella lista, perciò parte da 1, vista la presenza della parentesi. Il ciclo va avanti finché *indice* non si trova sull'elemento che verrebbe restituito dalla prossima next() e questo conteggio viene fatto da *i*.

```
StringTokenizer st = new StringTokenizer(list," ,",false);
int indice = 1;
int i = 0;
while(st.hasMoreTokens() && i < li.nextIndex()) {
    int gap = st.nextToken().length();
    indice = indice + gap + 3;
    i++;
}
```

Sistemata la posizione di *indice*, il cursore, in una situazione normale, viene posto a *indice* – 2, per posizionarlo dopo l'elemento appena processato e prima dell'elemento restituito dalla prossima next(). Ovviamente va controllato anche che indice non sfori le estremità della lista e, in quel caso, appendere il cursore alla fine con ris.append('^') o posizionarlo prima della lista.

Flavio Maiorana – Matricola 182611