Progetto#2 LinkedList – Corso di Programmazione Orientata agli Oggetti

Mattia Gatto

Matricola:182926

II anno Ingegneria Informatica

**Traccia**

Progetto#2 LinkedList – Corso di Programmazione Orientata agli Oggetti

È assegnata la seguente interfaccia List<T> appartenente al package poo.util, che si ispira liberamente al collection framework di Java (package java.util), mantenendo per i metodi omonimi lo stesso significato, tipi di eccezioni etc.

package poo.util; public interface List<T> extends Iterable<T>{ int size(); void clear(); boolean contains( T e ); boolean isEmpty(); void add( T e ); void remove( T e ); ListIterator<T> listIterator(); ListIterator<T> listIterator( int pos ); void addFirst( T e ); void addLast( T e ); T removeFirst(); T removeLast(); T getFirst(); T getLast(); void sort( Comparator<T> c ); }//List

Prima parte Sfruttando la possibilità dei metodi default e metodi statici (consentiti da Java 8 e versioni successive) nelle interfacce, concretizzare, con l’aiuto degli iteratori, quanti più metodi è possibile direttamente nell’interfaccia List<T>. Metodi che non possono essere realizzati in List<T>, concretizzarli, ove possibile, come parte dello sviluppo di una classe astratta AbstractList<T> che implementa List<T> ed espone certamente una realizzazione dei metodi toString(), equals() e hashCode(). Sviluppare quindi la classe concreta LinkedList<T> che estende AbstractList<T> e memorizza gli elementi su una lista concatenata a doppio puntatore. Nel progetto di LinkedList<T> rilevante è la realizzazione delle strutture di iterazione (iteratore semplice e list iterator). Si ricorda che un iteratore, quando non è al di fuori della lista, si trova logicamente tra due elementi consecutivi e non su un elemento. Al fini di identificare correttamente, quando esiste, l’elemento corrente (definito da previous() o next()), può essere opportuno introdurre nella classe dell’iteratore una coppia di puntatori <previous, next> in mezzo ai quali si colloca logicamente l’iteratore. Inoltre una enumerazione può essere utilmente introdotta per esprimere i possibili movimenti in avanti (FORWARD), indietro (BACKWARD) o l’assenza di movimento (UNKNOWN), e mantenere in una variabile di istanza l’ultimo movimento effettuato con l’iteratore. Poiché Java non consente di dichiarare un tipo enum in una inner class, collocare l’enumerazione direttamente nell’outer class LinkedList. Si nota che dopo un movimento in avanti (next()), l’elemento corrente è riferito dal puntatore previous; dopo un movimento indietro (previous()) l’elemento corrente è riferito dal puntatore next. Siccome un ListIterator è un Iterator, limitarsi ad implementare una sola inner class per il ListIterator<T>. Un oggetto di tale classe, a questo punto, è restituibile sia dal metodo iterator() (di Iterable<T>) che dai metodi listIterator([pos]). Quando si restituisce un oggetto list iterator mediante iterator(), esso verrà utilizzato con i soli tre metodi hasNext(), next() e remove() come prescritto dal tipo statico Iterator<T> etc.

Il metodo sort() riceve un oggetto Comparator<T> e ordina la lista this in accordo all’algoritmo bubble sort.

Il codice Java ottenuto va testato accuratamente. Si suggerisce di predisporre una classe Main col metodo main che verifica le “varie” condizioni di funzionamento. Nei casi di dubbio, confrontarsi con l’uso di java.util.LinkedList<T>. Come altro test, provare a rimpiazzare nella classe PolinomioLL l’uso della LinkedList di Java, con le classi/interfacce parte di questo progetto.

Inizialmente ho creato una cartella(javaProgect)denominata ProgettoPOO2 all’interno della quale ho creato due package: uno (poo.progetto2) dove ho creato una classe “Test” per verificare tutti i metodi di LinkedList; un secondo (poo.util) dove ho creato le classi interfaccia , astratta e concreta della LinkedList.

Nel package poo.util nella intefaccia List:

**package** poo.util;

**import** java.util.Comparator;

**import** java.util.Iterator;

**import** java.util.ListIterator;

**import** java.util.NoSuchElementException;

Inizialmente ho denominato il package nel quale ci troviamo, successivamente ho utilizzato gli import di Comparator,Iterator , ListIterator e NoSuchElementException per implementare in default i metodi sotto creati nell’ interfaccia.

Dichiariamo l’interfaccia con il nome “List” di tipo generico T e la estendiamo Iterable di tipo generico T.

**public** **interface** List<T> **extends** Iterable<T>{

**default** **int** size() {

**int** c=0;

Iterator<T> it= iterator();

**while** (it.hasNext())c++;

**return** c;

}

Nel metodo chiamato ‘size’ che deve restituire un intero andiamo a creare una variabile di tipo intero ‘c’ e la inizializziamo a zero, successivamente andiamo a prendere l’iteratore che chiameremo ’it’ con la funzione Iterator<T> it= iterator().  
Successivamente diciamo che fino a quanto it.hasNext(), ossia, se l iteratore ‘it’ trova un altro elemento successivo facciamo c++, quindi incrementiamo il contatore, e alla fine restituiamo il contatore ‘c’.

**default** **void** clear() {

Iterator<T> it= iterator();

**while** (it.hasNext()) {it.next();it.remove();}

}

Nel metodo chiamato ‘clear’ che è void quindi non ha niente da restituire andremo a prendere l’iteratore che chiameremo ’it’ con la funzione Iterator<T> it= iterator() e diremo che fino a quanto it.hasNext(), ossia se l iteratore ‘it’ trova un altro elemento successivo, facciamo it.next ossia prendiamo il prossimo elemento e lo rimuoviamo con it.remove().

**default** **boolean** contains( T e ) {

Iterator<T> it= iterator();

**while** (it.hasNext())

**if**(it.next().equals(e))**return** **true**;

**return** **false**;

}

Nel metodo chiamato ‘contains’ che deve restituire un valore booleano e ha come paramento un valore ‘e’ di tipo T generico andiamo a prendere l’iteratore che chiameremo ’it’ con la funzione Iterator<T> it= iterator() e diremo che fino a quanto it.hasNext(), ossia se l’iteratore ‘it’ trova un altro elemento successivo facciamo it.next, ossia prendiamo il prossimo elemento e vedremo se è uguale a ‘e’ in tal caso ritorna true , altrimenti false.

**default** **boolean** isEmpty() {

**return** !iterator().hasNext();

}

Nel metodo chiamato ‘isEmpty’ che deve restituire un valore booleano ritorniamo il valore booleano dato dall’equivalenza !iterator().hasNext()(ossia se l’iteratore non ha elementi successivi);

//void add(T e);

**default** **void** add(T e) {

ListIterator<T> lit= listIterator();

**while**(lit.hasNext()) lit.next();

lit.add(e);

}

Nel metodo chiamato ‘add’ che è void quindi non ha niente da restituire e ha come parametro un generico elemento di tipo T chiamato ‘e’, andremo a creare un ListIterator di tipo T chiamato ‘lit’ con la funzione ListIterator<T> lit= listIterator() e diremo che fino a quanto il ListIterator ‘lit’ ha altri elementi prossimi facciamo lit.next(), per prendere l’elemento successivo(per far scorrere il ListIterator fino a fine lista) e usciti dal while facciamo lit.add(e) per aggiungere il generico elemento ‘e’ alla lista.

**default** **void** remove( T e ) {

Iterator<T> it= iterator();

**if**(!it.hasNext()) **throw** **new** NoSuchElementException();

**while**(it.hasNext()) {

**if**(it.next().equals(e)) {

it.remove();

**break**;

}

}

}

Nel metodo chiamato ‘remove’ che è void quindi non ha niente da restituire e ha come parametro un generico elemento di tipo T chiamato ‘e’, andiamo a prendere l’iteratore che chiameremo ’it’ con la funzione Iterator<T> it= iterator() e diremo che se!it.hasNext() ossia se l’ iteratore ‘it’ non ha altri elementi allora lanciamo una nuova eccezione con la funzione **throw** **new** NoSuchElementException()(che va a definire che non ci sono altri elementi), altirmenti diremo che fino a quanto it.hasNext(), ossia se l’iteratore ‘it’ trova un altro elemento successivo, facciamo it.next ossia prendiamo il prossimo elemento e vedremo se è uguale a ‘e’, in tal caso facciamo it.remove() per eliminare l’ elemento e usciamo dal ciclo con la funzione break, altrimenti se non lo trova non fa niente.

ListIterator<T> listIterator();

ListIterator<T> listIterator( **int** pos );

Questi sono due metodi che andremo a definire nella classe concreta LinkedList;

//void addFirst(T e);

**default** **void** addFirst( T e ) {

ListIterator<T> lit=listIterator();

lit.add(e);

}

Nel metodo chiamato ‘addFirst’ che è void quindi non ha niente da restituire e ha come parametro un generico elemento di tipo T chiamato ‘e’, andremo a creare un ListIterator di tipo T chiamato ‘lit’ con la funzione ListIterator<T> lit= listIterator e facciamo lit.add(e) per aggiungere il generico elemento ‘e’ alla lista( in modo tale da inserirlo all’inizio della lista scandita dal nuovo iteratore).

//void addLast(T e);

**default** **void** addLast(T e) {

ListIterator<T> lit= listIterator();

**while**(lit.hasNext()) lit.next();

lit.add(e);

}

Nel metodo chiamato ‘addLast’ che è void quindi non ha niente da restituire e ha come parametro un generico elemento di tipo T chiamato ‘e’, andremo a creare un ListIterator di tipo T chiamato ‘lit’ con la funzione ListIterator<T> lit= listIterator() e diremo che fino a quanto il ListIterator ‘lit’ ha altri elementi prossimi, facciamo lit.next() per prendere l’elemento successivo(per far scorrere il ListIterator fino a fine lista) e usciti dal while facciamo lit.add(e) per aggiungere il generico elemento ‘e’ alla lista.(è uguale al metodo add).

**default** T removeFirst() {

**if**(isEmpty()) **throw** **new** NoSuchElementException("La lista è vuota");

Iterator<T> it= iterator();

T l=it.next();

it.remove();

**return** l;

}

Nel metodo chiamato ‘remove’ che è void quindi non ha niente da restituire, andiamo a dire che se isEmpty() ossia se non ha altri elementi allora lanciamo una nuova eccezione con la funzione **throw** **new** NoSuchElementException()(che va a definire che non ci sono altri elementi), altrimenti andiamo a prendere l’iteratore che chiameremo ’it’ con la funzione Iterator<T> it= iterator() , facciamo it.next ossia prendiamo il prossimo elemento e lo salviamo in una variabile ‘l’ di tipo generico T e facciamo it.remove() per eliminare l’ elemento in prima posizione nella lista. Alla fine ritorniamo ‘l’.

**default** T removeLast() {

**if**(isEmpty()) **throw** **new** NoSuchElementException();

Iterator<T> it= iterator();

T l=**null**;

**while** (it.hasNext())l=it.next();

it.remove();

**return** l;

}

Nel metodo chiamato ‘remove’ che è void quindi non ha niente da restituire, andiamo a dire che se isEmpty() ossia se non ha altri elementi allora lanciamo una nuova eccezione con la funzione **throw** **new** NoSuchElementException()(che va a definire che non ci sono altri elementi), altrimenti andiamo a prendere l’iteratore che chiameremo ’it’ con la funzione Iterator<T> it= iterator() ,creiamo una variabile ‘l’ di tipo generico T e la inizializziamo a null, in seguito diremo che fino a quanto it.hasNext(), ossia ‘it’ ha altri elementi facciamo it.next, quindi prendiamo il prossimo elemento e lo salviamo in ‘l’ e usciti dal while alla fine del ciclo facciamo it.remove() per eliminare l’ elemento in ultima posizione nella lista. Alla fine ritorniamo ‘l’.

**default** T getFirst() {

**if**(isEmpty()) **throw** **new** NoSuchElementException();

Iterator<T> it= iterator();

T l=it.next();

**return** l;

}

Nel metodo chiamato ‘getFirst’ che è di tipo T quindi deve restituire un elemento di tale tipo generico, andiamo a dire che se isEmpty() ossia se non ha altri elementi allora lanciamo una nuova eccezione con la funzione throw new NoSuchElementException()(che va a definire che non ci sono altri elementi), altrimenti andiamo a prendere l’iteratore che chiameremo ’it’ con la funzione Iterator<T> it= iterator() ,creiamo una variabile ‘l’ di tipo generico T e in questa andiamo a inserire il prossimo elemento della lista con la funzione it.next() e alla fine restituiamo ‘l’.(per restituire il primo elemento della lista).

**default** T getLast() {

**if**(isEmpty()) **throw** **new** NoSuchElementException();

Iterator<T> it= iterator();

T l=**null**;

**while** (it.hasNext())l=it.next();

**return** l;

}

Nel metodo chiamato ‘getLast’ che è di tipo T quindi deve restituire un elemento di tale tipo generico, andiamo a dire che se isEmpty() ossia se non ha altri elementi allora lanciamo una nuova eccezione con la funzione throw new NoSuchElementException()(che va a definire che non ci sono altri elementi), altrimenti andiamo a prendere l’iteratore che chiameremo ’it’ con la funzione Iterator<T> it= iterator() ,creiamo una variabile ‘l’ di tipo generico T e la inizializiamo a null, in seguito diremo che fino a quanto it.hasNext(), ossia ‘it’ ha altri elementi facciamo it.next, quindi prendiamo il prossimo elemento e lo salviamo in ‘l’ e usciti dal while alla fine del ciclo restituiamo ‘l’.(per restituire l’ ultimo elemento della lista).

**void** sort( Comparator<T> c );

Questo metodo andrà definito nella classe concreta LinkedList.

}//List

Qui finisce l’ interfaccia List.

**package** poo.util;

**import** java.util.Iterator;

Dopo aver definito il package di destinazione e aver importato l’iterator andiamo a creare un AbstractList che è una classe astratta che è di tipo <T> generico e implementa l’interfaccia List di tipo generico <T>.

**public** **abstract** **class** AbstractList<T> **implements** List<T> {

**public** **boolean** equals(Object o) {

**if**(!(o **instanceof** List)) **return** **false**;

**if** (**this**==o)**return** **true**;

List <T> l=(List)o;

**if**(l.size()!=**this**.size())**return** **false**;

Iterator <T>it1= **this**.iterator();

Iterator <T>it2= l.iterator();

**while**(it1.hasNext()) {

**if**(!it1.next().equals(it2.next()))**return** **false**;

}

**return** **true**;

}

Nel metodo chiamato ‘equals’ che deve restituire un valore booleano e riceve come parametro un Oggetto chiamato ‘o’ andremo inizialmente a dire che se l’oggetto ‘o’ non è di tipo List con l’istruzione (!(o instanceof List)) allora ritorniamo false, poi vediamo se il this ossia l’ oggetto chiamante è uguale ad ’o’ allora ritorna true, altrimenti creiamo una List di tipo generico T che chiameremo ‘l’ e in questi ci mettiamo ‘o’ castizzato a List con la seguente riga di comando: List <T> l=(List)o; successivamente vediamo che se la dimensione di ‘l’ con la funzione l.size è diversa dalla dimensione di this allora ritorniamo false, altrimenti prendiamo un Iterator <T> chiamato ‘it1’ che sarà uguale a this.iterator()(ossia all’ iteratore di this) e prendiamo un Iterator <T> chiamato ‘it2’ che sarà uguale a l.iterator()(ossia all’ iteratore di ‘l’) e diremo che fino a quanto it1.hasNext() perché avendo lo stesso numero di elementi ora dobbiamo controllare se sono gli stessi elementi nelle stesse posizioni, quindi vediamo se l’ elemento successivo di ‘it1’ è diverso dall’ elemento successivo di’it2’ in tal caso ritorna false, alla fine del ciclo usciti fuori ritorniamo true( perché se non è mai entrato in false allora le due liste saranno uguali).

**public** **int** hashCode() {

**final** **int** M=43;

**int** h=0;

**for**(T o:**this**) {

h=h\*M+o.hashCode();

}

**return** h;

}

Nel metodo chiamato ”hashCode” che deve restituire un intero andiamo inizialmente a creare una variabile intera final(ossia non modificabile) chimata “M” a cui assegniamo il valore di 43(anche se si può assegnare un valore di un qualsiasi numero primo)e un intero chiamato ‘h’ che inizializziamo a 0 e successivamente diciamo che per ogni elemento ‘o’ di tipo generico in this, ossia sul chiamante h=h\* M

+ l’haschcode di ‘o’. Alla fine dopo essere usciti dal for ritorniamo ‘h’.

**public** String toString() {

Iterator <T>it= **this**.iterator();

StringBuilder sb= **new** StringBuilder(500);

sb.append("[ ");

**while**(it.hasNext()){

sb.append(it.next());

**if**(it.hasNext())sb.append(',');

}

sb.append(" ]");

**return** sb.toString();

}

}

Nel metodo chiamato ‘toString’ che deve restituire una Stringa, andiamo a prendere l’iteratore che chiameremo ’it’ con la funzione Iterator<T> it= this.iterator()(sarà quindi uguale all’ iteratore del chiamante) e creiamo un nuovo StringBuilder chiamato ’sb’ di dimensione 500 e appendiamo una parentesi quadra aperta all’ inizio, successivamente diciamo che fino a quanto it.hasNext(), quindi fino a quanto l’ iteratore non ha più elementi prossimi, allora appendiamo ad ‘sb’ il prossimo elemento dell’ iteratore e verifichiamo che se l’ iteratore ha ancora un altro elemento allora appendiamo anche una virgola , alla fine del ciclo appendiamo una parentesi quadra e ritorniamo il metodo toString di ‘sb’.

Ora che abbiamo concluso anche la classe astratta andiamo nella concreta chiamata LinkedList:

**package** poo.util;

**import** java.util.Comparator;

**import** java.util.Iterator;

**import** java.util.ListIterator;

**import** java.util.NoSuchElementException;

Dopo aver impostato il package nel quale ci troviamo e aver impostato gli import dovuti andiamo creare la classe LinkedList di tipo generico T che estende AbstractList di tipo generico T.

**public** **class** LinkedList<T> **extends** AbstractList<T> {

**public** **enum** Movement{***BACKWARD***,***FORWARD***,***UNKNOWN***};

**public** **int** size = 0;

**private** Nodo<T> testa = **null**, coda = **null**;

//nodo iniziale e finale della LinkedList

**private** **static** **class** Nodo<E>{

E info; //informazione

Nodo<E> prior, next; //nodi adiacenti

}

Inizialmente ho creato una enumerazione pubblica chiamata ‘Movement’ che contiene i seguenti movimenti: BACKWARD,FORWARD,UNKNOWN.

Successivamente ho creato una variabile pubblica chiamata size che ho inizializzato a zero perché all’inizio è sempre zero, e due variabili private di tipo Nodo generico in E chiamate ‘testa’ e ‘coda’ che ho inizializzato a null e rappresentano il primo e l’ultimo elemento. Inoltre qui abbiamo creato una inner class chiamata Nodo di tipo generico E che ha un’informazione di tipo generica E chiamata ‘info’ e i nodi adiacenti di tipo Nodo generici in E chiamati ‘prior’ e ‘next’ ossia precedente e successivo.

**public** **int** size(){

**return** size;

}

Questo è un metodo size che ritorna la dimensione della Lista;

**public** **void** add(T e){

**this**.addLast(e);

}//add

Questo un metodo add che ha un parametro ‘e’ generico in T che restituisce il metodo addLast(T e) perché sono entrambi uguali.

**public** **void** remove(T e){

**if**(!**this**.contains(e))

**return**;

ListIteratore l = **new** ListIteratore(0);

**boolean** flag = **false**;

**while**(!flag){

T x = l.next();//portiamo l'iteratore avanti

**if**(x.equals(e)){

l.remove();//rimozione

flag = **true**;//trovato

}//if

}//while

}//remove

Qui ho ri-concretizzato il metodo “remove” che inizialmente controlla se l’ elemento c’è o non c’è nella lista grazie al metodo contains , se non c’è infatti c’è un return senza niente, mentre altrimenti creiamo un listIteratore ‘l’ in posizione 0, e una variabile ‘flag’ che è booleana e la inizializziamo a false, e diciamo che fino a quanto !flag(diversa da flag, ossia fino a quanto flag non è =true) impostiamo un elemento di tipo generico in T che chiameremo ‘x’ a cui gli attribuiamo il valore di l.next, ossia il prossimo elemento del ListIteratore, e verifichiamo se ‘x’=’e’ allora l’ abbiamo trovato e quindi facciamo l.remove() e poi impostiamo flag =true.

**public** **void** addFirst(T e) {

Nodo<T> n = **new** Nodo<>(); n.info = e;

n.next = testa;

n.prior = **null**;

**if**(isEmpty()) //se la lista è vuota

coda = n; //n sarà anche il last

**else**{

testa.prior = n;

}

testa = n; //first punterà ad n

size++;

} // addFirst

Qui ho ri-concretizzato il metodo “addFirst”, che è void e riceve come parametro un elemento generico in T chiamato ’e’. Inizialmente creiamo un nuovo nodo con la funzione Nodo<T> n = new Nodo<>() e gli attribuiamo come info ‘e’ e successivamente impostiamo n.next= testa e n.prior = null, ossia l’ elemento nodo successivo sarà sicuro ‘testa’ mentre l’ elemento nodo precedente sarà null, ora se la lista è vuota allora coda=n perché il nostro nodo sarà anche l’ ultimo elemento, altrimenti il precedente del nodo testa sarà il nuovo nodo ‘n’, alla fine impostiamo che la testa della lista sarà ‘n’ e incrementiamo la size.

**public** **void** addLast(T e) {

Nodo<T> n = **new** Nodo<>(); n.info = e;

n.next = **null**;

n.prior = coda;

**if**(isEmpty())//se la lista è vuota

testa = n;//n sarà anche il last

**else**{

coda.next = n;

}

coda = n; // last punterà a n

size++;

}//addLast

Qui ho ri-concretizzato il metodo “addLast”, che è void e riceve come parametro un elemento generico in T chiamato ’e’. Inizialmente creiamo un nuovo nodo con la funzione Nodo<T> n = new Nodo<>() e gli attribuiamo come info ‘e’ e successivamente impostiamo n.next= null e n.prior = coda ossia l’ elemento nodo successivo sarà sicuro ‘null’ mentre l’ elemento nodo precedente sarà ‘coda’, ora se la lista è vuota allora testa=n perché il nostro nodo sarà anche l’ ultimo elemento, altrimenti il successivo del nodo coda sarà il nuovo nodo ‘n’, alla fine impostiamo che la coda della lista sarà ‘n’ e incrementiamo la size.

**public** T removeFirst() {

**if**(testa == **null**) **throw** **new** NoSuchElementException();

T x = testa.info;//elemtno da rimuovere

**if**(testa == coda){ //lista fatta da un solo nodo

coda = **null**; testa = **null**;

}

**else**{//lista fatta da più di 1 nodo.

testa = testa.next;

testa.prior = **null**;

}

size--;

**return** x;

}//removeFirst

Qui ho ri-concretizzato il metodo “removeFirst” che è di tipo generico T quindi deve restituire un elemento generico in T.Inizialmente andiamo a controllare se la testa = null ossia se la lista è vuota in tal caso lanciamo una nuova eccezione con la funzione throw new NoSuchElementException()(che va a definire che non ci sono altri elementi), altrimenti andiamo a creare un elemento generico in T che chiamiamo ‘x’ e al quale assegniamo l’informazione della testa, ora se la testa è uguale alla coda impostiamo tutte e due a null perché vuol dire che la lista è formata da un solo nodo, altrimenti vuol dire che la lista ha più di un elemento e impostiamo che la testa sarà il nodo successivo a testa e il nodo precedente a testa sarà impostato a null. Alla fine ritorniamo ‘x’ e decrementiamo la size.

**public** T removeLast() {

**if**(coda == **null**) **throw** **new** NoSuchElementException();

T x = coda.info; //elemento da rimuovere

**if**(coda == testa){//lista fatta da un solo nodo

coda = **null**; testa = **null**;

}

**else**{//lista fatta da più di 1 nodo

coda = coda.prior;

coda.next = **null**;;

}

size--;

**return** x;

}//removeLast

Qui ho ri-concretizzato il metodo “removeFirst” che è di tipo generico T quindi deve restituire un elemento generico in T. Inizialmente andiamo a controllare se la coda= null ossia se la lista è vuota in tal caso lanciamo una nuova eccezione con la funzione throw new NoSuchElementException()(che va a definire che non ci sono altri elementi), altrimenti andiamo a creare un elemento generico in T che chiamiamo ‘x’ e al quale assegniamo l’informazione della coda, ora se la testa è uguale alla coda impostiamo tutte e due a null perché vuol dire che la lista è formata da un solo nodo, altrimenti vuol dire che la lista ha più di un elemento e impostiamo che la coda sarà il nodo precedente a coda e il nodo successivo a coda sarà impostato a null. Alla fine ritorniamo ‘x’ e decrementiamo la size.

@Override

**public** T getFirst() {

**if**(isEmpty()) **throw** **new** NoSuchElementException();

**return** testa.info;

}//getFirst

Qui ho ri-concretizzato il metodo “getFirst” che è di tipo generico T quindi deve restituire un elemento generico in T. Inizialmente andiamo a controllare se isEmpty ossia se la lista è vuota in tal caso lanciamo una nuova eccezione con la funzione throw new NoSuchElementException()(che va a definire che non ci sono altri elementi), altrimenti andiamo a ritornare l’informazione della testa.

@Override

**public** T getLast() {

**if**(isEmpty()) **throw** **new** NoSuchElementException();

**return** coda.info;

}//getLast

Qui ho ri-concretizzato il metodo “getLast” che è di tipo generico T quindi deve restituire un elemento generico in T. Inizialmente andiamo a controllare se isEmpty ossia se la lista è vuota in tal caso lanciamo una nuova eccezione con la funzione throw new NoSuchElementException()(che va a definire che non ci sono altri elementi), altrimenti andiamo a ritornare l’informazione della coda.

**public** ListIterator<T> listIterator(){

**return** **new** ListIteratore(0);

} //Iteratore della LinkedList

Questo metodo chiamato “listIterator” va a restiruire un ListIterator in posizione iniziale 0.

**public** Iterator<T> iterator(){

**return** **new** ListIteratore(0);

}//Iteratore Semplice

Questo metodo chiamato “Iterator” va a restiruire un ListIterator in posizione iniziale 0.

**public** ListIterator<T> listIterator(**int** pos){

**return** **new** ListIteratore(pos);

}//Iteratore della LinkedList partendo da pos

Questo metodo chiamato “listIterator” che ha come parametro un intero chiamato ‘pos’ va a restituire un ListIterator in posizione iniziale ‘pos’.

**private** **class** ListIteratore **implements** ListIterator<T>{

**private** Nodo<T> previous = **null**,corrente = testa;

//il cursore si trova tra 2 nodi

**private** Movement movimento = Movement.***UNKNOWN***;

//descrive l'ultimo movimento effettuato

//(se è forward,si è spostato in avanti)

//(se è backward,si è spostato indietro)

Questa è una inner class chiamata’ ListIteratore’che implementa ListIterator di tipo generico in T, che inizialmente crea due Nodi privati che sono previous che inizializza a null e corrente che inizializza a testa, il cursore quindi va a trovarsi fra i due nodi, in più va a dichiarare un Movement chiamato ‘movimento’ ad *UNKNOWN ossia descrive l’ultimo movimento effettuato e in questo momento ancora è nullo.*

**public** ListIteratore(**int** pos){

**if**(pos < 0 || pos > size()) **throw** **new** IllegalArgumentException ("Posizione illegale");

**int** i = 0;

**while**(i < pos){

**if**(pos == size()-1){

previous = coda;

corrente = **null**;

}

previous = corrente;

corrente = corrente.next;

i++;

}

}//costruttore

Questo è un costruttore del ListIteratore che riceve come parametro un intero chiamato ‘pos’ e inizialmente va a verificare se ‘pos’ è minore di zero o è maggiore della dimensione della lista in tal caso solleva l’ eccezione IllegalArgumentException che vuol dire che abbiamo dato un argomento sbagliato perché non si può trovare nella posizione data, altrimenti chiamiamo una variabile intera ‘i’ e la inizializiamo a zero che ci servirà come contatore, e diciamo che fino a quanto ‘i’ è minore di ‘pos’ allora: se ‘pos’ è = size -1 ossia se la posizione e alla fine della lista allora ‘previous’ sarà la coda e il ‘corrente’ sarà null;, altrimenti il ‘previous’ sarà uguale al ‘corrente’ e il nostro ‘corrente’ sarà uguale al nodo successivo al ‘corrente’ , e per concludere incrementiamo ‘i’.

**public** **boolean** hasNext(){

**return** corrente != **null**;

}

//hasNext

Il metodo ”hasNext” va a ritornare un booleano che corrisponderà alla verifica se il ‘corrente’ è diverso da null, ossia l’ elemento successivo deve essere diverso da null;

**public** T next(){

**if**(!hasNext()) **throw** **new** NoSuchElementException();

previous = corrente;

corrente = corrente.next;

movimento = Movement.***FORWARD***;

**return** previous.info;

}

//next

Il metodo ”Next” che restituisce un elemento generico in T verifica inizialmente se è diverso da hasnext (cioè se non ha elementi prossimi) in questo caso lancia l’ eccezione NoSuchElementException( cioè che non riconosce nessun’ altro elemento).  
Altrimenti ‘previous’ sarà uguale al ‘corrente’ , il ‘corrente’ sarà uguale al nodo successivo al ‘corrente’(al corrente.next), e il movimento viene settato a FORWARD(avanti) e facciamo ritornare l’informazione del ‘previous’.

**public** **boolean** hasPrevious(){

**return** previous != **null**;

}

//hasPrevious

Il metodo ” hasPrevious” va a ritornare un booleano che corrisponderà alla verifica se il ‘previous’ è diverso da null ossia l’ elemento precedente deve essere diverso da null;

**public** T previous(){

**if**(!(hasPrevious())) **throw** **new** NoSuchElementException();

corrente = previous;

previous = previous.prior;

movimento = Movement.***BACKWARD***;

**return** corrente.info;

}

//previous

Il metodo ” previous” che restiruisce un elemento generico in T verifica inizialmente se è diverso da hasPrevious(cioè se non ha elementi pprecedenti) in questo caso lancia l’ eccezione NoSuchElementException( cioè che non riconosce nessun’ altro elemento).  
Altrimenti ‘corrente’ sarà uguale al ‘previous’ , il ‘previous’ sarà uguale al nodo precedente al ‘previous’(al previous.prior), e il movimento viene settato a *BACKWARD* (indietro) e facciamo ritornare l’informazione del ‘corrente’.

**public** **void** add(T e) {

Nodo<T> a = **new** Nodo<>();

a.info = e;

**if**(testa == **null** && coda == **null**){//lista vuota

testa = a;//unico nodo

coda = a;//unico nodo

a.prior = **null**;

a.next = **null**;

}

**else** **if**(previous == **null**){//cursore a 0

testa.prior = a;

a.prior = **null**;

a.next = testa;

testa = a;

}

**else** **if**(corrente == **null**){//cursore a size

a.next = **null**;

a.prior = coda;

coda.next = a;

coda = a;

}

**else**{//cursore con una lista con più di 2 elementi

a.next = corrente;

a.prior = previous;

corrente.prior = a;

previous.next = a;

}

previous = a;

size++;

}//add

Il metodo “add” che ha un parametro ‘e’ generico in T ed è void, va a creare inizialmente un nuovo nodo che chiama ‘a’ a cui gli attribuisce informazione ‘e’ e successivamente controlla se:

* La testa e la coda sono a null, ossia che la lista è vuota, la testa diventerà ‘a’ , la coda’ sarà ‘a’, e i precedenti e successivi di ‘a’ saranno settati a null;
* Previous è uguale a null, ossia il puntatore si trova a zero, allora il precedente della testa sarà ‘a’, il precedente di’a’ sarà null e il suo successivo sarà testa, e impostiamo la testa ad ‘a’;
* Corrente è uguale a null, il puntatore si trova a size, quindi portiamo il successivo di ‘a’ a null e il suo precedente a coda, il successivo della coda diventerà ‘a’, e impostiamo la coda ad ‘a’;
* Altrimenti il cursore si trova in una lista con più di due elementi e non si trova ne all’inizio e ne alla fine, quindi il successivo di ‘a’ sarà corrente e il suo precedente sarà previous, il precedente del nodo corrente sarà ‘a’ e il successivo del nodo previous sarà ‘a’;

Alla fine impostiamo il previous ad ‘a’ e incrementiamo la size.

**public** **int** nextIndex() {

**int** i = 0;

Nodo<T> prossimo = testa;

**boolean** flag = **false**;

**while**(!flag && prossimo!=**null**){

**if**(prossimo.equals(corrente)){

flag = **true**;//trovato

**return** i;

}

**else**{

i++;

prossimo = prossimo.next;

}

}

**return** -1;

}

//nextIndex

Il metodo” nextIndex()” deve restituire un intero , inizialmente dichiariamo un intero ‘i’ uguale a 0 , un nodo ‘prossimo’ che sarà la testa inizialmente, e un booleano chiamato ’flag’ che è inizializzato a false, successivamente diciamo che fino a quanto flag non è true e ‘prossimo’non è null, allora vediamo se il nodo ’prossimo’ è uguale al nodo ’corrente’ in tal caso impostiamo flag a true e ritorniamo ’i’(che sarà l’indice), altrimenti incrementiamo ‘i’ e impostiamo come ‘prossimo’ il successivo a ‘prossimo’, usciti dal while se non l’abbiamo trovato ritorniamo -1.

**public** **int** previousIndex() {

**int** i = 0;

Nodo<T> prossimo = testa;

**boolean** flag = **false**;

**while**(!flag && prossimo!=**null**){

**if**(prossimo.equals(previous)){

flag = **true**;

**return** i;

}

**else**{

i++;

prossimo = prossimo.next;

}

}

**return** -1;

}//previousIndex

Il metodo” nextIndex()” deve restituire un intero , inizialmente dichiariamo un intero ‘i’ uguale a 0 , un nodo ‘prossimo’ che sarà la testa inizialmente, e un booleano chiamato ’flag’ che è inizializzato a false, successivamente diciamo che fino a quanto flag non è true e ‘prossimo’ non è null, allora vediamo se il nodo ’prossimo’ è uguale al nodo ’previous’ in tal caso impostiamo flag a true e ritorniamo ’i’(che sarà l’indice), altrimenti incrementiamo ‘i’ e impostiamo come ‘prossimo’ il successivo a ‘prossimo’, usciti dal while se non l’ abbiamo trovato ritorniamo -1.

**public** **void** remove() {

**if**(movimento == Movement.***UNKNOWN***) **throw** **new** IllegalStateException();

Nodo<T> n = **null**;

**if**(movimento == Movement.***BACKWARD***)

n = corrente;

**else**

n = previous;

//in n abbiamo il nodo da rimuovere

**if**(n == testa){

testa = testa.next;

**if**(testa == **null**)

coda = **null**;

**else**

testa.prior = **null**;

}

**else** **if**(n == coda){

coda = coda.prior;

coda.next=**null**;

}

**else**{

n.prior.next = n.next;

n.next.prior = n.prior;

}

**if**(movimento == Movement.***BACKWARD***)

corrente = n.next;

**else**

previous = n.prior;

movimento = Movement.***UNKNOWN***;

size--;

}

//remove

Il metodo “remove” che è void, va a verificare inizialmente se il movimento è settato ad UNKNOWN in tal caso lanciamo l’ eccezione IllegalStateException, altrimenti va a creare un nuovo nodo ‘n’ che a cui assegna null e successivamente controlla se:

* Il movimento è uguale a BACKWARD, in tal caso ‘n’ sarà il ‘corrente’, altrimenti ‘n’ sarà il ‘previous’, in ‘n’ così avremo il nodo da rimuovere;
* Se ‘n’ è uguale a testa, allora la testa sarà il nodo successivo a testa, ma se questa nuova testa è uguale a null, allora impostiamo la coda a null, altrimenti impostiamo l’elemento precedente a testa a null;
* Se ‘n’ è uguale a coda, allora la coda sarà il nodo precedente a coda e il successivo di coda sarà null;
* Altrimenti impostiamo che il nodo precedente a ‘n’ come successivo punterà al successivo di ‘n’, il nodo successivo a ‘n’ come precedente punterà al precedente di ’n’;
* Ora se il movimento è uguale a BACKWARD allora il nodo corrente sarà il successivo a ‘n’, altrimenti il nodo previous sarà il precedente a ‘n’;

Alla fine impostiamo il movimento ad UNKNOWN e decrementiamo la size.

**public** **void** set(T e) {

**if**(movimento == Movement.***UNKNOWN***)//nessun spostamento

**throw** **new** IllegalStateException();

**if**(movimento == Movement.***BACKWARD***)

corrente.info = e;

**else**

previous.info = e;

}//set

Questo metodo chiamato ”set” che è void quindi non ha ritorni ed ha come parametro un tipo generico in T chiamato ‘e’ inizialmente vede se il movimento è impostato ad *UNKNOWN in tal caso non ci sono stati spostamenti e lancia l’ eccezione di stato illegale (*IllegalStateException), se invece movimento è impostato a *BACKWARD allora impostiamo che l’ informazione del corrente sarà uguale ad ‘e’ , altrimenti l’ informazione del previous sarà impostata ad ‘e’(perché il movimento è impostato a FORWARD).*

}

**public** **void** sort(Comparator<T> c) {

Nodo<T> a = testa;

**int** j = size - 1;

**while**(j > 0){

**int** i = 0;

**for**(;i<j; i++){

**if**(c.compare(a.info, a.next.info)>0){

T temp = a.info;

a.info = a.next.info;

a.next.info = temp;

}

a = a.next;

}

a = testa;

j--;

}

}

}

Questo metodo chiamato ”Sort” che è void quindi non ha ritorni ed ha come parametro un Comparator che è di tipo generico in T chiamato ‘c’, inizialmente crea un nuovo nodo ‘a’ e lo mette = a testa, prende un intero ‘j’ e lo mette = a size -1, e poi indichiamo che fino a quando j è maggiore di 0 allora impostiamo un intero ‘i’ a 0 e facciamo un for, in modo tale che per ogni elemento tale che ‘i’ è minore di ‘j’ ,incrementiamo ‘j’, ora se il confronto dato da ‘c’ fra l‘informazione di ‘a’ e l’informazione del successivo di ‘a’ è maggiore di zero, (allora vuol dire che il primo è maggiore del secondo) quindi utilizziamo una variabile temporanea generica in T per effettuare lo scambio chiamata ’tmp’, e mettiamo in ‘tmp’ l’informazione di ‘a’, l’informazione di ‘a’ diventerà l’informazione del successivo di ‘a’ e l’ informazione del successivo di ‘a’ diventerà ‘tmp’, altrimenti ‘a’ sarà uguale a testa. Infine usciti dal for decrementiamo j. ( questo è un bubleSort per ordinare una LinkedList).

Finita anche questa ultima classe concreta nel’ package poo.progetto2 ho usato una classe chamata “Testt” per verificare se tutti i metodi sopra espressi sono corretti:

**package** poo.progetto2;

**import** java.util.ListIterator;

**import** poo.util.\*;

**public** **class** Test {

**public** **static** **void** main(String[] args){

List<Integer> ll=**new** LinkedList<>();

System.***out***.println("Test \*1\* addLast di <9,2,5>");

ll.add(9); ll.addLast(2); ll.addLast(5);

System.***out***.println(ll);

System.***out***.println("Test \*2\* addFirst di <7,8,4>");

ll.addFirst(7); ll.addFirst(8); ll.addFirst(4);

System.***out***.println(ll);

System.***out***.println("Test \*3\* tre removeFirst");

ll.removeFirst(); ll.removeFirst(); ll.removeFirst();

System.***out***.println(ll);

System.***out***.println("Test \*4\* sorting");

ll.sort((x,y)->{ **return** x-y; } );

System.***out***.println(ll);

System.***out***.println("Test \*5\* add di <-1,-2> in testa - su listIterator()");

ListIterator<Integer> lit=ll.listIterator();

lit.add(-1); lit.add(-2);

System.***out***.println(ll);

System.***out***.println("Test \*6\* ripresa iteratore forward dopo cascata di add in testa");

**while**( lit.hasNext() ){

System.***out***.print( lit.next()+" ");

}

System.***out***.println();

System.***out***.println("Test \*7\* ripresa iteratore backward dopo forward");

**while**( lit.hasPrevious() ){

System.***out***.print( lit.previous()+" ");

}

System.***out***.println("Test \*8\* size");

System.***out***.println("size="+ll.size());

System.***out***.println("Test \*9\* add di <30,40> in coda - su listIterator(size)");

lit=ll.listIterator(ll.size());

lit.add(30); lit.add(40);

System.***out***.println(ll);

System.***out***.println("Test \*10\* add di <100,500> partire dall'indice 3 - su listIterator(3)");

lit=ll.listIterator(3);

lit.add(100); lit.add(500);

**try**{

lit.remove();

}**catch**( IllegalStateException e ){

System.***out***.println("Errore: remove proibita dopo una add");

}

System.***out***.println(ll);

System.***out***.println("Test \*11\* rimozione elemento");

lit=ll.listIterator();

**while**( lit.hasNext() ){

**int** x=lit.next();

**if**( x==500 ) lit.remove();

}

System.***out***.println(ll);

System.***out***.println("Test \*12\* re-sorting");

ll.sort((x,y)->{ **return** x-y; } );

System.***out***.println( ll );

System.***out***.println("Test \*13\* insertion sort di <57,23,12> - forward");

**int** []a={57,23,12};

**for**( **int** x: a){

**boolean** flag=**false**;

ListIterator<Integer> it=ll.listIterator();

**while**( it.hasNext() && !flag ){

**int** y=it.next();

**if**( y>=x ){

it.previous();

it.add(x);

flag=**true**;

}

}

**if**( !flag ) it.add(x);

}

System.***out***.println(ll);

System.***out***.println("Test \*14\* rimozione di <57,12,23>");

ListIterator<Integer> li=ll.listIterator();

**while**( li.hasNext() ){

**int** u=li.next();

**if**( u==12 || u==23 || u==57 ){

li.remove();

**try**{

li.remove();

}**catch**( IllegalStateException e ){

System.***out***.println("Errore: remove proibita dopo remove");

}

}

}

System.***out***.println(ll);

System.***out***.println("Test \*15\* insertion sort di <57,12,23> - backward");

**for**( **int** x: a){

**boolean** flag=**false**;

ListIterator<Integer> it=ll.listIterator(ll.size());

**while**( it.hasPrevious() && !flag ){

**int** y=it.previous();

**if**( y<=x ){

it.next();

it.add(x);

flag=**true**;

}

}

**if**( !flag ) it.add(x);

}

System.***out***.println(ll);

System.***out***.println("Test \*16\* clear e sort subito dopo");

ll.clear();

ll.sort((x,y)->{**return** x-y;});

System.***out***.println(ll);

System.***out***.println("Test \*17\* sort");

List<Integer> lin=**new** LinkedList<>();

lin.addFirst(9); lin.addFirst(2); lin.addFirst(-3); lin.addFirst(4); lin.addFirst(1);

System.***out***.println("Lista lin iniziale="+lin);

lin.sort((x,y)->{**return** x-y;});

System.***out***.println("Lista lin sorted="+lin);

System.***out***.println("Bye.");

}//main

}//Test