Esercizio: Liste, utilizzando una classe astratta ricorsiva

Introduzione. Abbiamo già visto nelle lezioni di teoria (Lezione 15 nelle dispense) come rappresentare liste di oggetti usando la classe MiniLinkedList con:

- attributo first = ”l’indirizzo del primo nodo nella lista”
- attributo first=null se la lista è vuota.

Un’altra possibilità è rappresentare liste vuote e non vuote con oggetti con etichetta “lista vuota” e “lista non vuota”. Rappresentiamo in Java queste etichette con la classe delle liste vuote e la classe delle liste non vuote. Questa rappresentazione contiene delle informazioni in più, utili a Java per scegliere quale versione di un metodo usare attraverso il dynamic binding. Otteniamo questa rappresentazione considerando le liste come una classe astratta ricorsiva.

Nota: *non è essenziale usare le classi astratte ricorsive nel caso delle liste: usiamo le liste solo per avere un esempio semplice.*

Definiamo una classe astratta List delle liste *ordinate senza ripetizioni di interi*.

List ha due sottoclassi concrete:

- la classe Nil che contiene (oltre a null che NON usiamo in questa particolare rappresentazione per rappresentare le lista vuota) gli oggetti che rappresentano le liste vuote.
- la classe Cons degli oggetti che rappresentano liste contenenti un elemento “elem” e una lista “next”, fatta di elementi tutti più grandi di elem (oppure vuota).

Dunque Cons è definito a partire da List che è definito a partire da Cons: la definizione di List e Cons è ricorsiva. Possiamo usare List per rappresentare insiemi finiti di interi. Abbiamo scelto di usare liste ordinate senza ripetizioni: questa scelta ha il vantaggio di avere una sola rappresentazione per ogni insieme di valori.

Il primo passo è scegliere quali metodi implementare (tutti dinamici e pubblici). Chiediamo di avere:

(i) un metodo boolean empty() per decidere se una lista è vuota,

(ii) un metodo int size() per contare gli elementi di una lista,

(iii) un metodo boolean contains(int x) per decidere se una lista contiene un elemento x, e due metodi per costruire nuove liste ordinate senza ripetizioni. Innanzitutto il metodo

(iv) List insert(int x), che crea una nuova lista aggiungendo un elemento x a una lista data, dopo gli elementi più piccoli e prima di quelli più grandi, e non aggiunge x se x c'è già.

(v) List append(List l), che costruisce una nuova lista unione della lista data e della lista l.

(vi) nelle classi Nil e Cons riscriviamo il metodo String toString() per liste.

In questo esercizio non definiamo metodi che modificano liste già esistenti: questa scelta ha il vantaggio di evitare il rischio di sovrascrivere dati che ancora ci servono, e lo svantaggio di usare tempo e memoria in più, per costruire le nuove liste. Sfruttiamo il più possibile le liste esistenti, come vedremo.

Il costruttore Cons(int elem, List next), della classe Cons di liste non vuote, consente di definire liste non ordinate: per questo motivo non ne consentiamo l'uso fuori di List, della sua cartella e delle sue sottoclassi, e lo dichiariamo protected. Non possiamo definire Cons come private, perché Cons viene usato nella classe Nil e dunque fuori di Cons, anche se nella stessa cartella. La classe Nil ha il costruttore pubblico Nil() per la lista vuota.

Un'ultima scelta di implementazione è la seguente. Quando definiamo una nuova lista (ordinata e senza ripetizioni), vogliamo riutilizzare per quanto possibile la lista da cui partiamo. Siano

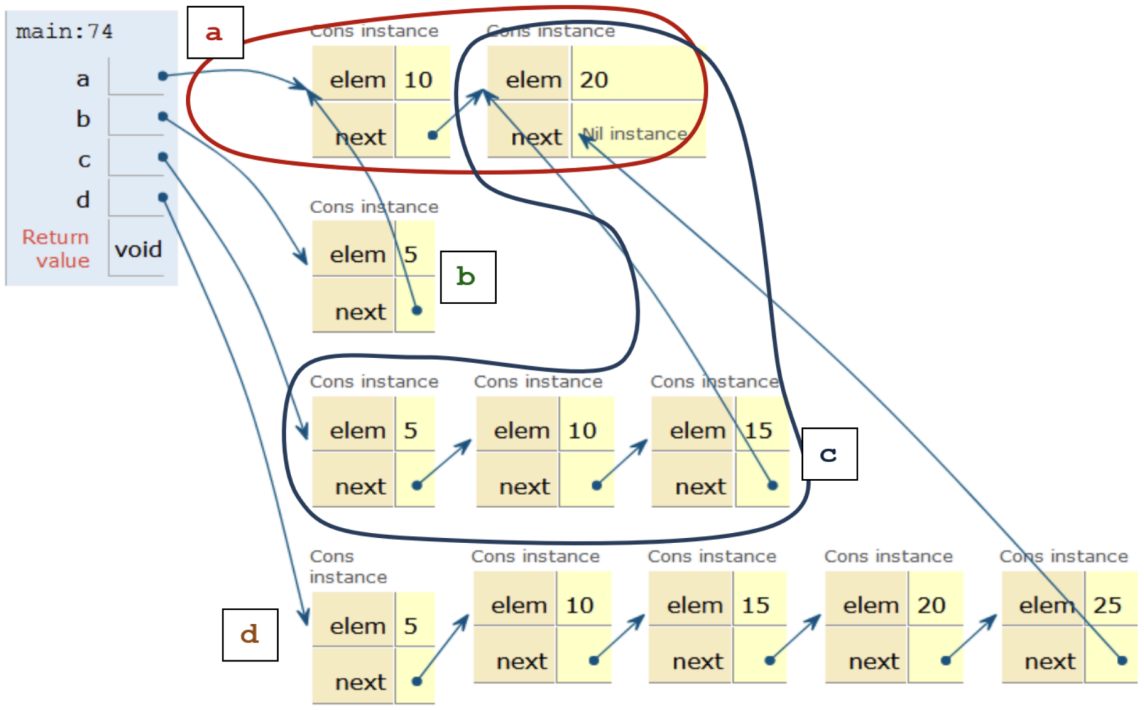
a = {10,20}

b = a.insert(5) = {5,10,20}

c = b.insert(15) = {5,10,15,20}

d = c.insert(25) = {5,10,15,20,25}

Vogliamo che insert costruisca le liste b,c,d riutilizzando per quando possibile le liste precedenti. Con la particolare definizione di List che sceglieremo, otterremo il seguente risultato: la lista b riutilizza tutti gli elementi di a, la lista c riutilizza l'elemento 20 di a e b, mentre la lista d riutilizza la sola costante Nil() che indica la lista vuota al fondo di a,b,c. Ecco il disegno: mostra che le liste a,b,c,d di cui sopra si sovrappongono in parte (ovvero condividono alcuni nodi).



[In questa cartella trovate la definizione di List e delle sue sottoclassi]

Esercizio 1 (rappresentazione alternativa per le liste). Si consideri la seguente classe astratta che modifica la classe **List.java** vista a lezione con l'introduzione di alcuni nuovi metodi astratti:

```
public abstract class List
{
  public abstract boolean empty();
  public abstract int size();
  public abstract boolean contains(int x);
  public abstract List insert(int x);
  public abstract List append(List l);
  // NUOVI METODI
  public abstract int sum();
  public abstract int get(int i);
  public abstract List succ();
  public abstract List filter_le(int x);
  public abstract List filter_gt(int x);
  public abstract List intersect(List l);
}
```

per rappresentare liste (vuote o non vuote) di numeri interi ordinati. In ciascuna istanza di **List**, gli interi compaiono in ordine crescente, e non ci sono mai duplicati. Modificare le

classi **Nil** e **Cons** viste a lezione per implementare i nuovi metodi di **List**. In particolare, dato un oggetto **p** di tipo **List**, deve essere possibile eseguire le seguenti operazioni:

• **p.sum()** ritorna la somma di tutti gli elementi della lista **p**.

• **p.get(i)** ritorna, se esiste, l'elemento che si trova all'indice **i** nella lista **p**. Al solito, il primo elemento di trova all'indice 0. Gestire in qualche modo il caso in cui **i** sia un indice non valido.

• **p.succ()** ritorna una nuova lista contenente tutti gli elementi di **p** incrementati di 1.

• **p.filter_le(x)** ritorna la sotto-lista di **p** che contiene tutti gli elementi di **p** minori o uguali a **x**. La lista ritornata è nuova.

• **p.filter_gt(x)** ritorna la sotto-lista di **p** che contiene tutti gli elementi di **p** maggiori di **x**. La lista ritornata è nuova.

• **p.intersect(l)** ritorna una nuova lista, intersezione di **p** e **l**.

Per verificare il corretto funzionamento dei metodi implementati è possibile eseguire il programma **TestList.java**.

Nota

• Tutti i metodi da implementare si scrivono in poche righe (molti in una sola riga, spesso corta) di codice. Se vi accorgete di scrivere codice più complicato, molto probabilmente la soluzione pensata non è quella ideale.