Descrizione

Seguenti task:

1. Implementare la classe `MyMultiset<E> implements Multiset<E>` tramite una efficiente rappresentazione degli elementi che non sprechi spazio
2. Implementare la classe `LinkedListDisjointSets` usando liste concatenate di elementi opportuni

Multiset

A differenza del concetto matematico di insieme, il multinsieme (multiset o bag in inglese) è un insieme in cui gli elementi possono essere inseriti più di una volta e il numero di occorrenze di ogni elemento è disponibile in ogni momento e può cambiare nel tempo a causa di inserimenti o cancellazioni. Tale numero, detto molteplicità o frequenza, non può essere negativo e se è zero significa che l'elemento non è presente nel multinsieme. Poiché la molteplicità è rappresentata con un `int`, un multinsieme non può mai contenere più di `Integer.MAX_VALUE` occorrenze per ogni elemento. Come nell'insieme classico, gli elementi di un multinsieme non sono indicizzati cioè non hanno una posizione specifica e non c'è un ordine particolare in cui gli elementi sono restituiti durante una iterazione.

L'interface `Multiset<E>` contiene la definizione delle API corrispondenti alle operazioni di base su un multinsieme di oggetti di un tipo generico `E`. In particolare, le API definite non permettono l'inserimento di elementi `null` e assumono che ci sia stata una ridefinizione opportuna dei metodi `boolean equals(Object o)` e `int hashCode()` nella classe `E` in modo da identificare univocamente un certo elemento tramite alcuni dei suoi campi.

Nell'implementazione, per rappresentare il multiset si può fare uso di altre interfacce o classi delle Collections della Java SE. Tuttavia la rappresentazione deve essere efficiente dal punto di vista dello spazio impiegato. Ciò significa che per allocare, per esempio, `Integer.MAX_VALUE` occorrenze di un certo elemento della classe `E` non si dovranno creare `Integer.MAX_VALUE` copie di puntatori all'elemento! Ciò infatti comporterebbe uno spreco enorme di spazio non necessario, poiché è sufficiente mantenere una sola copia dell'elemento e un solo puntatore allo stesso per poter rispondere in maniera corretta a tutte le operazioni richieste dalle API. Supponiamo ad esempio che per rappresentare `Integer.MAX_VALUE` occorrenze di uno stesso elemento si usino `Integer.MAX_VALUE` puntatori allo stesso. In una macchina con architettura a 64 bit in Java un puntatore occupa 8 byte e quindi si occuperebbero `8 * Integer.MAX_VALUE` bytes che sono circa **16 Gigabytes**!

Tra le API di `Multiset<E>` c'è anche il metodo `Iterator<E> iterator()`. Tale metodo deve restituire in iteratore per il multinsieme. Ciò significa che l'iteratore deve presentare tutti gli elementi del multinsieme (in un ordine qualsiasi perché gli elementi non sono indicizzati) e per ogni elemento deve presentare tutte le occorrenze. Inoltre, le occorrenze dello stesso elemento devono essere presentate in sequenza. L'iteratore restituito **non** deve implementare l'operazione `remove()`.

Ad esempio se un multinsieme `Multiset<Integer> m` è formato dai seguenti elementi:

2, 3, 4, 2, 1, -3, 2, -3

un iteratore `i` potrebbe restituire gli elementi nel seguente modo:

```
Iterator<Integer> i = m.iterator()
i.hasNext() == true
i.next() == 2
i.hasNext() == true
i.next() == 2
i.hasNext() == true
i.next() == 2
i.hasNext() == true
i.next() == 1
i.hasNext() == true
i.next() == -3
i.hasNext() == true
i.next() == -3
i.hasNext() == true
i.next() == 4
i.hasNext() == false
```

Si noti che le occorrenze dello stesso elemento, nell'esempio `2` e `-3`, sono sempre restituite in sequenza, mentre l'ordine degli elementi distinti è casuale.

L'iteratore restituito dall'implementazione deve essere **fail-fast**: se il multinsieme viene modificato strutturalmente (cioè viene fatta un'aggiunta o una cancellazione di almeno un'occorrenza di un elemento nuovo o già presente) in qualsiasi momento dopo la creazione dell'iteratore, l'iteratore dovrà lanciare una `java.util.ConcurrentModificationException` alla chiamata successiva del metodo `next()`. Se consideriamo l'esempio precedente:

```
Iterator<Integer> i = m.iterator()
i.hasNext() == true
i.next() == 2
i.hasNext() == true
i.next() == 2
i.hasNext() == true
i.next() == 2
i.hasNext() == true
i.next() == 1
i.hasNext() == true
i.next() == -3
m.add(9) <--- modifica strutturale al multiset
i.hasNext() == true
i.next() <--- lancia ConcurrentModificationException
```

Nel template fornito per la classe `MyMultiset<E>` sono da implementare tutti i metodi nei quali c'è il commento

```
// TODO implementare
```

e in generale tutti i punti contrassegnati con un commento `// TODO`.

Disjoint Sets con Liste Concatenate

Una collezione di insiemi disgiunti (disjoint sets) è una collezione di insiemi che hanno a due a due intersezione vuota, cioè un qualsiasi elemento o non appartiene alla collezione o, se appartiene, fa parte di un solo insieme tra quelli disgiunti attualmente esistenti. Se consideriamo una collezione di insiemi disgiunti di interi ad esempio possiamo avere:

- [ ] - la collezione vuota
- [{1}] - una collezione che contiene un solo insieme singoletto
- [{1}, {2}, {3}] - una collezione che contiene tre insiemi singoletto disgiunti
- [{1, 2}, {3}] - una collezione che contiene due insiemi disgiunti
- eccetera

In una collezione di insiemi disgiunti ogni insieme disgiunto ha, in ogni momento, un unico **rappresentante**, che è un elemento dell'insieme. Non è importante chi è il rappresentante, deve solo valere sempre la seguente **regola**: se si chiede il rappresentante di un insieme disgiunto due volte e, tra le due richieste, non è stata fatta nessuna modifica all'insieme stesso allora il rappresentante deve risultare essere lo stesso elemento. Se invece l'insieme viene modificato allora il rappresentante può cambiare.

Ci sono molti modi per rappresentare un insieme disgiunto, dai più semplici ai più complicati. La questione fondamentale che guida alla scelta della struttura dati da usare dipende, come succede spesso, delle operazioni che si vogliono implementare sulla struttura e dalla complessità computazionale che si desidera ottenere per esse. Nel caso delle collezioni di insiemi disgiunti si vogliono ottimizzare le seguenti operazioni:

- `makeSet(x)` - crea un nuovo insieme disgiunto singoletto che contiene il solo elemento `x` e `x` è anche il rappresentante dell'insieme
- `findSet(x)` - restituisce un puntatore all'elemento rappresentante dell'insieme disgiunto che attualmente contiene l'elemento `x`
- `union(x, y)` - unisce i due insiemi disgiunti che attualmente contengono l'elemento `x` e l'elemento `y`. Se `x` e `y` fanno già parte dello stesso insieme disgiunto allora l'operazione non fa niente. Se `x` e `y` fanno parte di insiemi diversi, diciamo `Sx` ed `Sy`, allora `Sx` ed `Sy` vengono eliminati dalla collezione e si inserisce un nuovo insieme disgiunto `Sxy` che contiene tutti gli elementi di `Sx` e di `Sy`. Il rappresentante del nuovo insieme `Sxy` è uno degli elementi di `Sx` o di `Sy`. L'implementazione può fornire indicazioni su chi sarà il rappresentante di `Sxy` oppure può lasciare il criterio di scelta non specificato.

L'interface `DisjointSets` fornisce le API per queste operazioni ed altre operazioni di base su collezioni di insiemi disgiunti.

Un esempio di uso delle operazioni principali è il seguente:

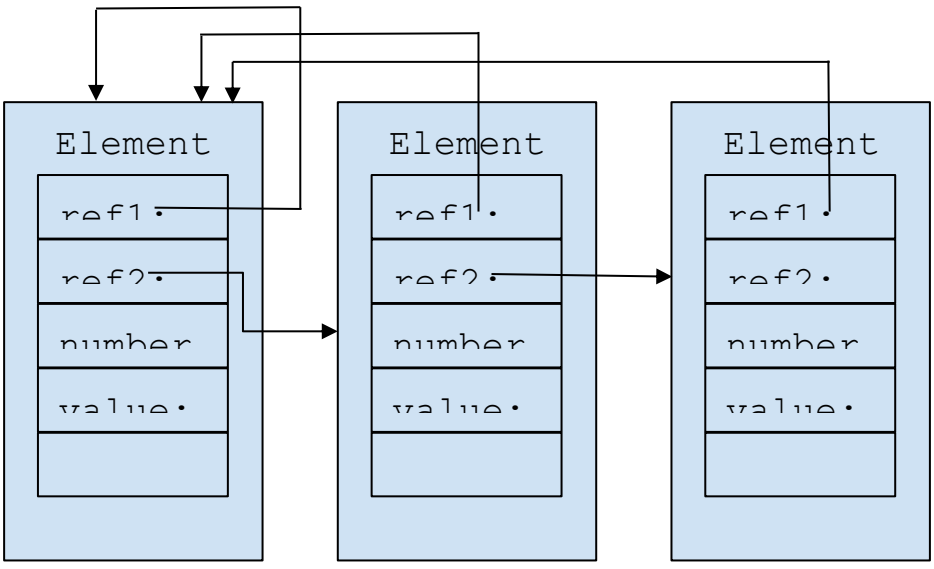
- all'inizio si parte da una collezione vuota [ ]
- `makeSet(3)` - si ottiene [{3}]

- `makeSet(5)` - si ottiene [{3}, {5}]
- `makeSet(7)` - si ottiene [{3}, {5}, {7}]
- `union(3,7)` - si ottiene [{3, 7}, {5}]
- `makeSet(1)` - si ottiene [{3, 7}, {5}, {1}]
- `makeSet(2)` - si ottiene [{3, 7}, {5}, {1}, {2}]
- `union(1,2)` - si ottiene [{3, 7}, {5}, {1, 2}]
- `union(1,7)` - si ottiene [{3, 7, 1, 2}, {5}]
- `union(3,5)` - si ottiene [{3, 7, 1, 2, 5}]

In questo progetto si deve implementare la collezione di insiemi disgiunti utilizzando **liste concatenate**. In questo tipo di rappresentazione un elemento di un insieme disgiunto deve avere, tra gli altri che caratterizzano la natura dell'elemento stesso (dati che rappresentano l'entità o l'attore nel dominio del discorso), i tre seguenti campi:

- un puntatore all'elemento rappresentante dell'insieme disgiunto di cui l'elemento fa parte. Se l'elemento è il rappresentante dell'insieme disgiunto di cui fa parte allora il puntatore punterà a se stesso - nel nostro caso questo puntatore si chiamerà genericamente `ref1`
- un puntatore al prossimo elemento nella lista concatenata che rappresenta l'insieme disgiunto di cui l'elemento fa parte - nel nostro caso questo puntatore si chiamerà genericamente `ref2`
- un intero che indica la dimensione attuale dell'insieme disgiunto. Questo valore deve essere settato solo presso l'elemento che è rappresentante dell'insieme disgiunto e può essere lasciato vuoto negli altri elementi - nel nostro caso questo puntatore si chiamerà genericamente `number`

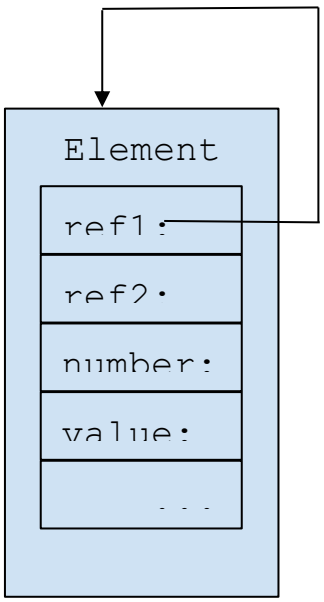
Graficamente possiamo visualizzare alcuni insiemi disgiunti di esempio come segue.



Insieme disgiunto {1, 4, 6} rappresentato da 4.

Il rappresentante (`value == 4`) è il primo elemento della lista concatenata e quindi il suo `ref1` punta a se stesso. Il campo `number` è settato alla dimensione dell'insieme che è 3. Il

puntatore `ref2` punta al prossimo elemento nella lista concatenata. L'elemento può contenere altri dati (...) che, per quanto riguarda la gestione degli insiemi disgiunti, non ci interessano. Non essendoci un ordine prestabilito il prossimo elemento è quello con `value == 1`. Qui il `ref1` punta all'elemento precedente che è il rappresentante dell'insieme disgiunto. Il campo `number`, non essendo lui il rappresentante, non è significativo. Il riferimento `ref2` punta al prossimo elemento della lista concatenata. Tale elemento è quello con `value == 6`. Anche qui il `ref1` punta all'elemento rappresentante dell'insieme disgiunto. Il campo `number`, non essendo lui il rappresentante, non è significativo. Il riferimento `ref2` è `null` il che indica che la lista concatenata termina qui e che non ci sono ulteriori elementi nell'insieme disgiunto.



Insieme disgiunto singoletto {2}, rappresentato da 2.

In questo caso la lista concatenata è formata da un solo elemento che è l'unico elemento (`value == 2`). Il `ref1` punta a se stesso, perché è lui il rappresentante e `ref2` è `null`. Il campo `number` è settato a 1.

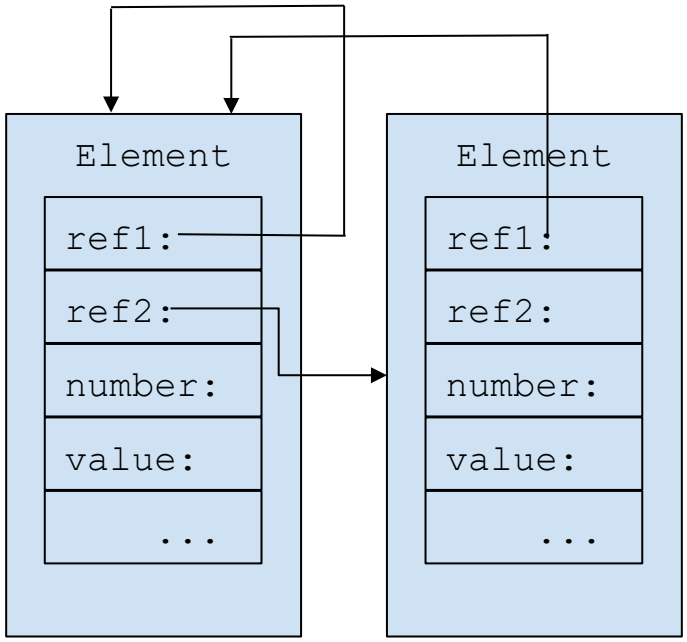
Con questa rappresentazione le operazioni di `makeSet(x)` e `findSet(x)` si possono realizzare in tempo costante Θ(1), se si ha un puntatore all'elemento `x`. L'implementazione fornita nella classe `LinkedListDisjointSets implements DisjointSets` deve fornire queste prestazioni. Ciò è possibile perché gli elementi degli insiemi disgiunti modellati da `DisjointSets` devono essere oggetti di una classe che implementa l'interface `DisjointSetElement` che richiede, appunto, a un oggetto di possedere i riferimenti `ref1` e `ref2` e il campo `number`. Il puntatore `x` quindi sarà sempre di tipo `DisjointSetElement` cioè sarà sempre una variabile polimorfa. Il tipo vero della classe che implementa l'interface non ci interessa. Un accorgimento fondamentale in questo contesto è che l'uguaglianza tra oggetti di tipo `DisjointSetElement` deve essere sempre testata utilizzando == cioè due oggetti `DisjointSetElement` sono uguali se e solo se sono fisicamente lo stesso oggetto. Tale accorgimento è dovuto al fatto che ciò che caratterizza l'elemento nell'insieme disgiunto sono proprio i valori dei puntatori `ref1` e `ref2`, che si aggiungono ai dati propri dell'oggetto (quelli della classe che implementa `DisjointSetElement`) e quindi il metodo `equals()` dell'oggetto non darebbe in generale

una nozione di uguaglianza corretta nel contesto degli insiemi disgiunti modellati come abbiamo fatto in questo progetto.

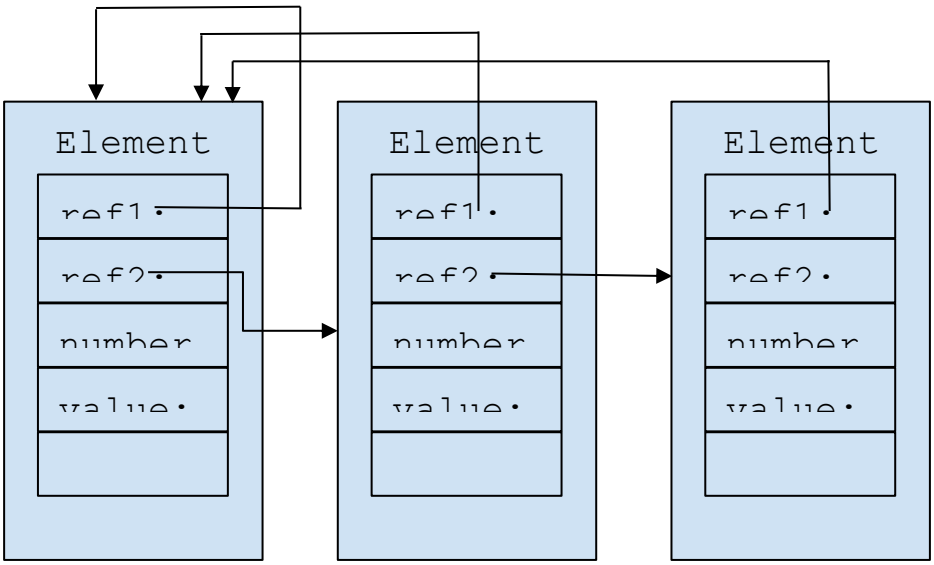
L'operazione di `union(x, y)` nella nostra implementazione con liste concatenate deve una complessità computazionale Θ(n) dove n = min(|`Sx`|,|`Sy`|) ovvero è il numero di elementi dell'insieme disgiunto più piccolo tra `Sx` e `Sy`. Ciò è possibile perché le liste concatenate possono essere unite in tempo costante e sono necessari n aggiornamenti del puntatore `ref1` degli elementi dell'insieme disgiunto più piccolo tra `Sx` ed `Sy`. Tale strategia deriva da ciò che è richiesto nella classe `LinkedListDisjointSets` a proposito del rappresentante dell'insieme unito (qui `x` è `e1` e `y` è `e2`):

```
/*
 * Dopo l'unione di due insiemi effettivamente disgiunti il rappresentante
 * dell'insieme unito è il rappresentante dell'insieme che aveva il numero
 * maggiore di elementi tra l'insieme di cui faceva parte {@code e1} e
 * l'insieme di cui faceva parte {@code e2}. Nel caso in cui entrambi gli
 * insiemi avevano lo stesso numero di elementi il rappresentante
 * dell'insieme unito è il rappresentante del vecchio insieme di cui faceva
 * parte {@code e1}.
 *
 * Questo comportamento è la risultante naturale di una strategia che
 * minimizza il numero di operazioni da fare per realizzare l'unione nel
 * caso di rappresentazione con liste concatenate.
 *
 */
@Override
public void union(DisjointSetElement e1, DisjointSetElement e2) { ...
```

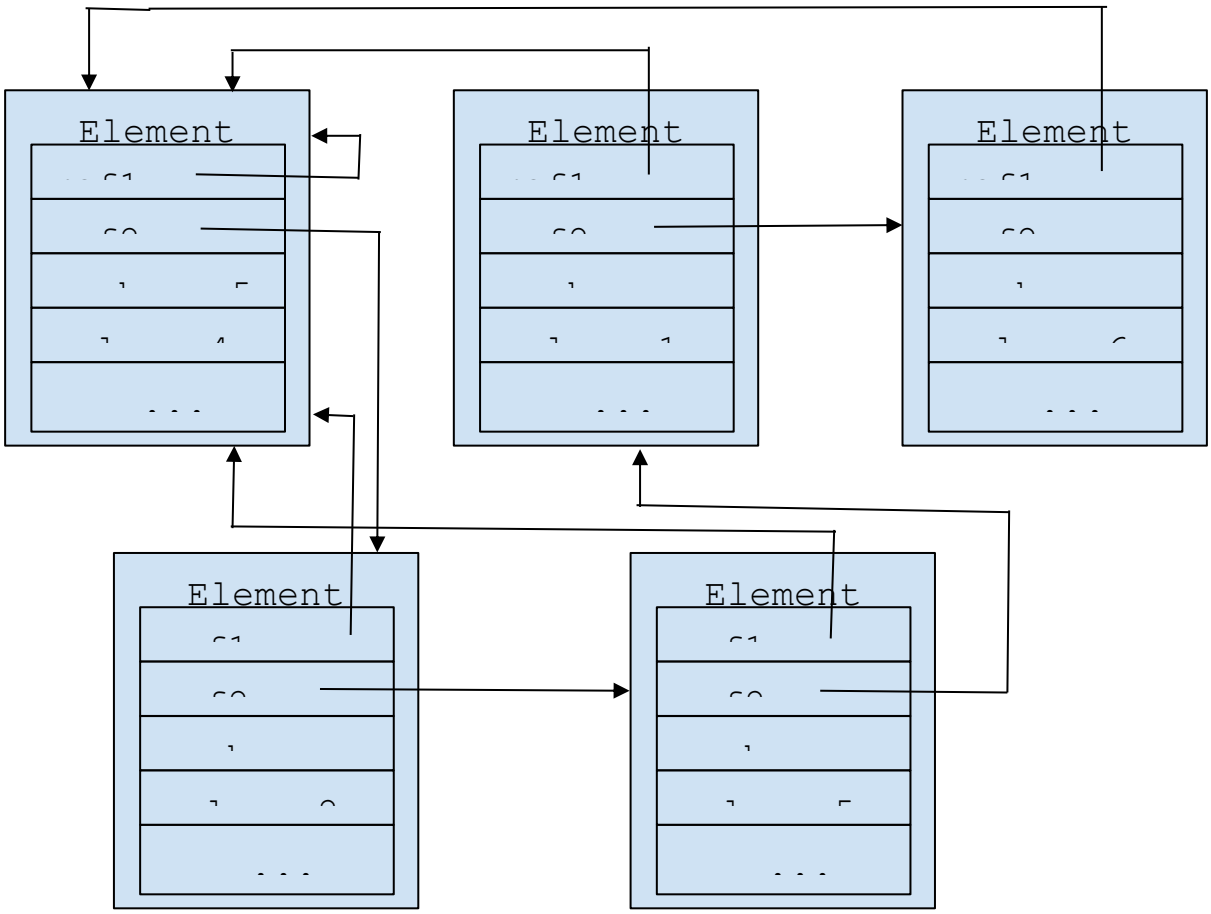
Ad esempio, se volessimo unire l'insieme `Sx` = {2, 5}, rappresentato da 2:



con l'insieme `Sy` = {1, 4, 6} rappresentato da 4:



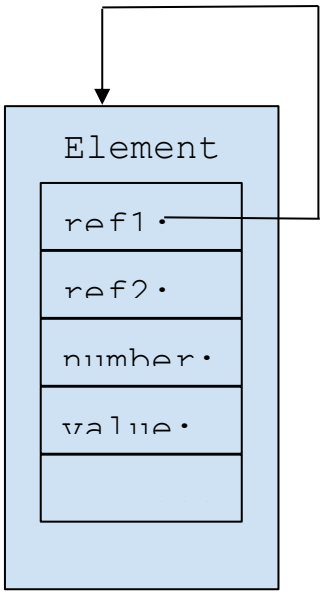
Il risultato dovrà essere il seguente:



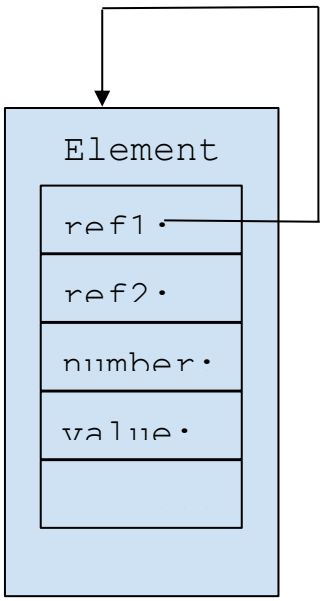
Cioè la lista corrispondente all'insieme più piccolo (in questo caso `Sx` = {2, 5}) è stata inserita tra il primo e il secondo elemento della lista corrispondente all'insieme più grande (in questo caso `Sy` = {1, 4, 6}) e sono stati aggiornati i riferimenti `ref1` dei due elementi con `value == 2` e `value == 5`. Il rappresentante del nuovo insieme {1, 2, 5, 4, 6} è 4, il vecchio rappresentante dell'insieme `Sy`, che era il più grande tra i due insiemi uniti. Per

realizzare l'unione è stato fatto l'inserimento della lista `Sx` nella lista `Sy` in posizione 1 che richiede tempo Θ(1) e sono stati fatti 2 aggiornamenti di puntatore degli elementi di `Sx` che sono 2 e quindi ha richiesto 2 operazioni. Infine si è aggiornato il campo `number` del vecchio (e nuovo) rappresentante di `Sy` con la somma del vecchio valore più il valore del vecchio rappresentante di `Sx`.

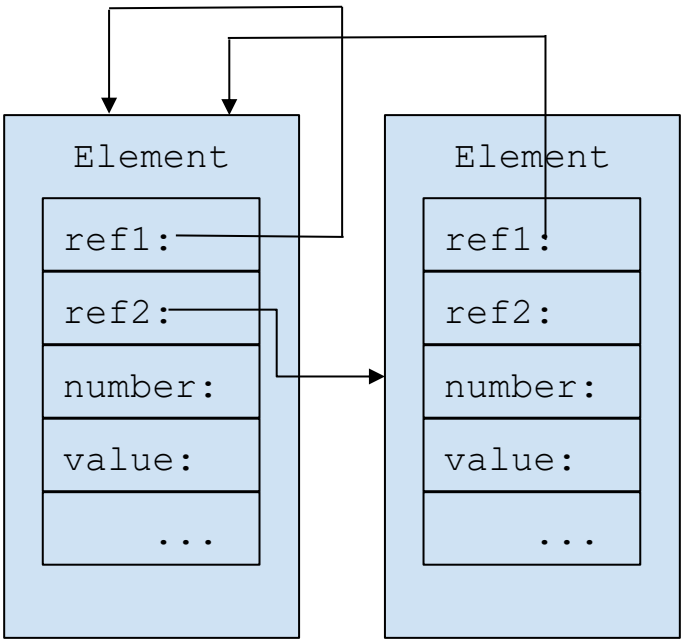
Nel caso in cui `Sx` e `Sy` abbiano lo stesso numero di elementi il nuovo rappresentante dovrà essere il vecchio rappresentante di `Sx`. Ad esempio, se dobbiamo unire `Sx` = {1}



con `Sy` = {2}



otterremo come risultato:



Abbiamo visto come rappresentare un singolo insieme disgiunto. Resta da specificare (e questo fa parte del task di implementazione della classe `LinkedListDisjointSets`) come rappresentare una collezione di insiemi disgiunti presenti in un certo momento all'interno di un oggetto della classe `LinkedListDisjointSets`. In questo caso si può utilizzare una opportuna struttura dati delle Collections di Java SE istanziando il tipo `E` con un tipo appropriato alla situazione. Anche in questo caso vanno seguiti i principi generali di efficienza, cioè utilizzare solo lo spazio necessario e usare strutture che minimizzano la complessità delle operazioni principali da effettuare.

E' infine fornita una classe `MyIntLinkedListDisjointSetElement implements DisjointSetElement` che rappresenta dei semplici elementi di un insieme disgiunto di numeri interi.