Multiset

Sezioni

Descrizione

Scopo della prova è progettare e implementare una gerarchia di oggetti utili a rappresentare dei *multiset*.

In matematica un **multiset** (detto anche *bag*) è una sorta di insieme in cui però, a differenza che in quest'ultimo, ciascun elemento può essere contenuto più di una volta; il numero di volte che un elemento compare in un multiset è detto **molteplicità** (di quell'elemento nel multiset).

Una conseguenza di questo fatto è che dati due elementi `a` e `b` si possono avere infiniti multiset distinti che li contengono:

- `{a, b}` contiene ciascun elemento con molteplicità 1 (e può essere quindi visto anche come insieme);
- in `{a, a, b}` l'elemento `a` ha molteplicità 2, mentre `b` ha ancora molteplicità 1;
- in `{a, a, a, b, b, b}`, sia `a` che `b` hanno molteplicità 3, e così via…

Come negli insiemi, ma diversamente dalle ennuple, o sequenze, l'ordine non conta così, ad esempio, `{a, a, b}` e `{a, b, a}` denotano lo stesso multiset.

La **cardinalità** di un multiset è la somma delle molteplicità dei suoi elementi; mentre il **supporto** di un multiset è l'insieme (senza ripetizioni) dei suoi elementi.

Molte operazioni e proprietà degli insiemi possono essere opportunamente definite per i multiset. In particolare, l'**unione** di due multiset `A` e `B` è il multiset `U` che ha per supporto l'unione dei supporti di `A` e `B` tale per cui la molteplicità di ciascuno elemento `u` in `U` è

Skip to main content

è il multiset `I` che ha per supporto l'intersezione dei supporti di `A` e `B` tale per cui la molteplicità di ciascuno elemento `u` in `U` è pari alla *minima* tra la molteplicità di `u` in `A` e in `B`

Cosa dovete implementare

Può scegliere se implementare un *multiset* di `String` oppure di tipo *generico* (maggiori dettagli a questo riguardo saranno dati in seguito).

Nel primo caso dovrà produrre almeno due implementazioni distinte dell'interfaccia

```
interface StringMultiSet extends Iterable<String> {
  int add(String s);
  int remove(String s);
  boolean contains(String s);
  int multiplicity(String s);
  int size();
  StringMultiSet union(StringMultiSet o);
  StringMultiSet intersection(StringMultiSet o);
}
```

dopo aver specificato nel dettaglio i vari metodi che, rispettivamente, devono:

- aggiungere un elemento a questo multiset, restituendo la molteplicità di tale elemento *dopo* l'inserimento;
- rimuovere un elemento da questo multiset, restituendo la molteplicità di tale elemento *prima* della rimozione (ignorando le richieste di rimuovere elementi non presenti nel multiset);
- restituire `true` se e solo se l'elemento specificato appartiene a questo multiset;
- restituire la *molteplicità* dell'elemento in questo multiset (restituendo 0 se l'elemento non appartiene al multiset);
- restituire la *cardinalità* di questo multiset;
- restituire il multiset ottenuto come *unione* di questo multiset con quello indicato come argomento (senza modificare questo multiset, o quello passato come argomento);
- restituire il multiset ottenuto come *intersezione* di questo multiset con quello indicato come argomenti (senza modificare questo multiset, o quello passato come argomento);

noti che inoltre il multiset deve poter iterare gli elementi del suo *supporto* (senza ripetizioni).

Le diverse implementazioni devono essere basate su rappresentazioni tra loro diverse del

Skip to main content

così via — o sulle strutture dati che riterrete più opportune).

L'efficienza delle implementazioni (in termini di costo computazionale in spazio e tempo) non costituirà elemento di valutazione, altrimenti detto, implementazioni poco efficienti *non* saranno penalizzate.

Non è richiesto l'*overriding* dei metodi `euqals` e `hashCode`, se nonostante questa indicazione decidesse di fornire una implementazione presti grande attenzione alla sua correttezza: implementazioni incomplete, o errate, saranno penalizzate nella valutazione!

Multiset generici

Al fine di migliorare la valutazione, se ha familiarità con i tipi generici, può scegliere di implementare una versione generica del *multiset* che soddisfi la seguente interfaccia (invece di quella specificata in precedenza):

```
interface MultiSet<E> extends Iterable<E> {
  int add(E e);
  int remove(Object o);
  default boolean contains(Object o);
  int multiplicity(Object o);
  int size();
  MultiSet<E> union(MultiSet<? extends E> o);
  MultiSet<E> intersection(MultiSet<? extends E> o);
}
```

i cui metodi, a meno del tipo `E` dell'elemento che in questo caso non è specificato in quanto parametrico, hanno lo stesso significato di quelli dell'interfaccia precedente.

Anche in questo caso, è richiesta l'implementazione di almeno due versioni basate su rappresentazioni distinte; è però fortemente sconsigliato procedere con entrambe le interfacce (provvedendo quindi almeno quattro implementazioni distinte): scelga da principio se seguire solo la strada del tipo concerto (*multiset* di stringhe) o generico.

La classe di test

Scrivete una classe che abbia un metodo statico `main` che legga dal flusso di ingresso due linee contenenti ciascuna una sequenza di parole separate da spazi e costruisca due multiset a partire da essi; una volta letto l'ingresso, la classe deve emettere sul flusso d'uscita, uno per linea, i due insiemi e la loro intersezione e unione, precedendo ogni insieme dall'indicazione della sua cardinalità

Skip to main content

Esempio

Leggendo dal flusso di ingresso

```
tre uno due uno tre tre
quattro due tre tre due
```

il programma emette

```
6 {tre: 3, uno: 2, due: 1}
5 {due: 2, tre: 2, quattro: 1}
8 {tre: 3, uno: 2, due: 2, quattro: 1}
3 {tre: 2, due: 1}
```

a meno dell'ordine in cui sono riportati gli elementi del supporto (ciascuno dei quali è seguito dalla sua molteplicità).

Soluzione

Data l'estrema semplicità del tema qui svilupperemo esclusivamente la soluzione basata sui *generici*; la soluzione per le sole stringhe può essere ottenuta a partire dal codice della presente a patto di fare qualche ovvia modifica (ad esempio, sostituendo il tipo parametrico `E` col tipo `String`).

L'interfaccia (e una classe astratta)

Riguardo alla documentazione dell'interfaccia (il cui codice è fornito nel testo), oltre a tradurre in commenti le specifiche informali del testo, dato che è assolutamente evidente che i metodi debbano essere *totali*, resta solo da occuparsi delle eventuali *eccezioni* che dovranno essere sollevate per i valori ritenuti "inaccettabili" come argomenti. Ha senso ispirarsi alle API del "Collections framework" con particolare riferimento all'interfaccia Set; come è facile constatare l'unica preoccupazione riguarda i riferimenti `null` nel metodo che aggiunge elementi all'insieme, in tutti gli altri casi non c'è bisogno di sollevare eccezioni; diverso il discorso per i metodi dell'unione ed intersezione che ovviamente devono sollevare eccezione se l'argomento è `null`.

Skip to main content

stessa come il metodo `multiplicity` e la capacità di iterare il proprio supporto. Per questa ragione è opportuno aggiungere le due implementazioni di *default* rispettivamente date da

```
default boolean contains(Object o) {
  return multiplicity(o) > 0;
}
```

[sorgente]

e

```
default int size() {
  int size = 0;
  for (E e : this) size += multiplicity(e);
  return size;
}
```

[sorgente]

Nello stesso spirito, il metodo `toString` è di ovvia implementazione a partire dai metodi dell'interfaccia; dato che è però impossibile sovrascrivere con un metodo di default un metodo di un supertipo (in questo caso, appunto il metodo `toString` di `Object`), può essere considerata l'introduzione di una *classe astratta* `AbstractMultiSet<E>` (senza stato) il cui unico obiettivo sia quello di fornire l'implementazione seguente

```
@Override
public String toString() {
  final StringBuilder sb = new StringBuilder("{");
  final Iterator<E> ie = iterator();
  while (ie.hasNext()) {
    final E e = ie.next();
    sb.append(e + ": " + multiplicity(e));
    if (ie.hasNext()) sb.append(", ");    }
  sb.append("}");
  return sb.toString();
}
```

[sorgente]

Si osservi per inciso che la cardinalità non è parte del risultato (ma sarà aggiunta nella soluzione), inoltre l'implementazione fa uso esplicito dell'iteratore (invece del costrutto *for-each*) al fine di poter stabilire se usare la virgola per separare gli elementi.

Skip to main content

Implementazione basata su una mappa

La prima e più ovvia rappresentazione di un multiset è data da una mappa che associ ciascun elemento con la sua molteplicità:

```
private final Map<E, Integer> elem2mult = new HashMap<>();
```

[sorgente]

Questa rappresentazione è molto efficiente sia in termini di *spazio* (ogni elemento è memorizzato una sola volta) che di *tempo* (usando l'implementazione HashMap le operazioni `get` e `put` hanno tempo costante).

L'*invariante di rappresentazione* (oltre alla banale richiesta che `elem2mult` non sia `null`, cosa garantita dall'inizializzazione e dal fatto che l'attributo è finale) è che la mappa non contenga chiavi `null` o valori non positivi (il che garantisce che il supporto del multiset coincida con l'insieme delle chiavi della mappa).

I due metodi mutazionali seguenti

```
@Override
public int add(E e) {
  Objects.requireNonNull(e);
  final int m = multiplicity(e);
  elem2mult.put(e, m + 1);
  return m + 1;
}

@Override
public int remove(Object o) {
  final int m = multiplicity(o);
  if (m == 1) elem2mult.remove(o);
  else if (m > 1) {
    @SuppressWarnings("unchecked")
    E e = (E) o;
    elem2mult.put(e, m - 1);
  }
  return m;
}
```

[sorgente]

sono di semplice implementazione ed è elementare osservare come rispettino l'invariante

Skip to main content

Altrettanto ovvia è l'implementazione del metodo che restituisce la molteplicità

```
@Override
public int multiplicity(Object o) {
  return elem2mult.containsKey(o) ? elem2mult.get(o) : 0;
}
```

[sorgente]

A questo punto può valere la pena di sovrascrivere i due metodi con implementazione di default nell'interfaccia; essi sono di fatto del tutto equivalenti

```
@Override
public boolean contains(Object o) {
  return elem2mult.containsKey(o);
}

@Override
public int size() {
  int size = 0;
  for (final int m : elem2mult.values()) size += m;
  return size;
}
```

[sorgente]

ma usano un accesso diretto alla mappa (non mediato da `multiplicity`), fatto che consente un minimo risparmio di tempo (evitando le chiamate intermedie).

Leggermente più sofisticata è l'implementazione dei metodi che hanno per argomento un altro multiset; iniziando dall'unione

```
@Override
public MultiSet<E> union(MultiSet<? extends E> o) {
  Objects.requireNonNull(o);
  HashMapMultiSet<E> result = new HashMapMultiSet<>();
  for (Map.Entry<E, Integer> elemMult : elem2mult.entrySet()) {
    final E elem = elemMult.getKey();
    result.elem2mult.put(elem, Math.max(elemMult.getValue(), o.multiplicity(elem)))
  }
  for (E elem : o)
    if (!elem2mult.containsKey(elem)) result.elem2mult.put(elem, o.multiplicity(ele
  return result;
}
```

[sorgente]

Skip to main content

osserviamo che il primo ciclo `for` aggiunge al risultato gli elementi del supporto di questo multiset (che è un sottoinsieme del supporto dell'unione), ma con la molteplicità massima tra quella in questo multiset e in quello passato per argomento (che sarà 0 se l'elemento non gli appartiene); il secondo `for` completa il risultato aggiungendo gli elementi del supporto dell'altro multiset non ancora aggiunti e con la loro molteplicità (che è la massima, dal momento che in questo multiset è 0, altrimenti sarebbero stati aggiunti nel primo ciclo).

L'intersezione

```
@Override
public MultiSet<E> intersection(MultiSet<? extends E> o) {
  Objects.requireNonNull(o);
  HashMapMultiSet<E> result = new HashMapMultiSet<>();
  for (Map.Entry<E, Integer> elemMult : elem2mult.entrySet()) {
    final E elem = elemMult.getKey();
    final int mult = Math.min(elemMult.getValue(), o.multiplicity(elem));
    if (mult > 0) result.elem2mult.put(elem, mult);
  }
  return result;
}
```

[sorgente]

usa un solo `for` sul supporto di questo multiset (che è un soprainsieme del supporto dell'intersezione) e calcola la molteplicità minima tra quella in questo multiset e in quello passato per argomento aggiungendo solo gli elementi per cui tale valore è positivo (ossia che sono contenuti anche nel supporto dell'altro multiset).

Per finire l'implementazione dell'iteratore può sfruttare il fatto osservato all'inizio che le chiavi della mappa sono di fatto il supporto del multiset

```
@Override
public Iterator<E> iterator() {
  return Collections.unmodifiableSet(elem2mult.keySet()).iterator();
}
```

[sorgente]

unico accorgimento, per non esporre la rappresentazione, è avvolgere l'insieme di chiavi con unmodifiableSet (altrimenti il metodo remove dell'iteratore potrebbe alterare le chiavi della mappa, dato che keySet) restituisce una vista).

Skip to main content

Implementazione basata su una lista

Una seconda possibilità è rappresentare un multiset tramite una lista con ripetizioni

```
  private final List<E> elems = new LinkedList<>();
```

[sorgente]

In questo caso l'*invariante di rappresentazione* (oltre alla solita richiesta che `elems` non sia `null`, cosa garantita dall'inizializzazione e dal fatto che l'attributo è finale) è che la lista non contenga elementi `null`.

I due metodi mutazionali seguenti

```
  @Override
  public int add(E e) {
    Objects.requireNonNull(e);
    elems.add(e);
    return multiplicity(e);
  }

  @Override
  public int remove(Object o) {
    final int m = multiplicity(o);
    elems.remove(o);
    return m;
  }
```

[sorgente]

sono di implementazione addirittura più semplice che nel caso precedente, dato che sfruttano gli omonimi metodi delle liste, (ed è ovvio che rispettino l'invariante).

L'implementazione del metodo che restituisce la molteplicità

```
  @Override
  public int multiplicity(Object o) {
    return Collections.frequency(elems, o);
  }
```

[sorgente]

qui fa uso di un metodo della classe di utilità Collections, ma potrebbe altrettanto

Skip to main content

In questo caso è necessario sovrascrivere i due metodi con implementazione di default nell'interfaccia

```
@Override
public boolean contains(Object o) {
  return elems.contains(o);
}

@Override
public int size() {
  return elems.size();
}
```

[sorgente]

il primo infatti consente di risparmiare tempo se l'elemento cercato è all'inizio della lista (`multiplicity` deve scandire tutta la lista per determinare il numero di occorrenze), ma è il secondo che fa davvero la differenza: questa implementazione richiede un tempo costante, mentre quella di default richiede tempo quadratico (una scansione della lista per determinare il supporto e una scansione per ciascun elemento per determinarne la molteplicità).

L'implementazione dei metodi che hanno per argomento un altro multiset anche in questo caso richiede un po' di attenzione; iniziando dall'unione

```
@Override
public MultiSet<E> union(MultiSet<? extends E> o) {
  Objects.requireNonNull(o);
  final ListMultiSet<E> result = new ListMultiSet<>();
  result.elems.addAll(elems);
  for (final E elem : o) {
    final int mult = o.multiplicity(elem) - multiplicity(elem);
    if (mult > 0) result.elems.addAll(Collections.nCopies(mult, elem));
  }
  return result;
}
```

[sorgente]

dapprima verranno aggiunti tutti gli elementi di questo multiset (usando addAll) con la loro molteplicità, quindi quelli del multiset passato per argomento, con la molteplicità eccedente a quella già aggiunta (che, nel caso non fossero presenti anche in questo multiset, è di fatto alla molteplicità nell'altro multiset); per aggiungere più copie di un elemento viene usato il metodo nCopies, ma lo stesso effetto potrebbe essere ottenuto tramite un ciclo `for`.

Skip to main content

```
@Override
public MultiSet<E> intersection(MultiSet<? extends E> o) {
  Objects.requireNonNull(o);
  final ListMultiSet<E> result = new ListMultiSet<>();
  for (final E elem : this) {
    final int mult = Math.min(multiplicity(elem), o.multiplicity(elem));
    if (mult > 0) result.elems.addAll(Collections.nCopies(mult, elem));
  }
  return result;
}
```

[sorgente]

ribadisce di fatto l'idea già presentata per la rappresentazione basata sulla mappa.

La complessità maggiore nell'uso di questa rappresentazione è l'implementazione dell'iteratore sul supporto del multiset. Non è infatti accettabile usare un Set per memorizzare il supporto, tale informazione è già presente nella lista e non ha senso duplicarla: vanificherebbe l'uso dell'astrazione iterazione!

La soluzione segue l'usuale approccio di utilizzare una classe anonima per realizzare l'iteratore

```
@Override
public Iterator<E> iterator() {
  return new Iterator<>() {

    private final Iterator<E> it = elems.iterator();
    private E next = null;
    private int idx = -1;

    @Override
    public boolean hasNext() {
      if (next != null) return true;
      while (it.hasNext()) {
        final E candidate = it.next();
        idx++;
        if (elems.indexOf(candidate) == idx) {
          next = candidate;
          return true;
        }
      }
      return false;
    }

    @Override
    public E next() {
      if (!hasNext()) throw new NoSuchElementException();
      final E result = next;
      next = null;
```

Skip to main content

```
        }
      };
    }
```

[sorgente]

la logica di “avanzamento” dell’iteratore è nel metodo evidenziato che mantiene traccia dell’indice `idx` dell’elemento iterato con `candidate` e lo “approva” come possibile `next` se la prima occorrenza dell’elemento candidato (calcolata tramite il metodo `indexOf`) coincide con `idx` (ossia se è la prima volta che viene visto).