Insiemi (capitolo 14.3 - la nostra trattazione è abbastanza diversa)

Department of Information Engineering

Il tipo di dati astratto Insieme (Set)

- È un contenitore (eventualmente vuoto) di oggetti **distinti** (cioè non contiene duplicati)
  - Senza alcun particolare ordinamento o memoria dell'ordine in cui gli oggetti sono inseriti/estratti
  - Corrisponde alla **nozione matematica di insieme**
- Definiamo la nostra astrazione di insieme tramite le seguenti operazioni
  - **inserimento** di un oggetto
    - fallisce silenziosamente se l'oggetto è già presente
  - **verifica della presenza** di un oggetto
  - **ispezione di tutti gli oggetti**
    - restituisce un array (in generale non ordinato) di riferimenti agli oggetti dell'insieme
  - **Non** definiamo un'operazione di **rimozione**
    - useremo l'operazione di sottrazione tra insiemi (cfr. più avanti)

Operazioni sugli insiemi

- Per due insiemi A e B, si definiscono le operazioni
  - **unione**, A ∪ B
    - appartengono all'unione di due insiemi tutti e soli gli oggetti che **appartengono ad almeno uno** dei due insiemi
  - **intersezione**, A ∩ B
    - appartengono all'intersezione di due insiemi tutti e soli gli oggetti che **appartengono a entrambi** gli insiemi
  - **sottrazione**, A - B (oppure anche A \ B)
    - appartengono all'insieme sottrazione tutti e soli gli oggetti che **appartengono ad A e non appartengono a B**
    - non è necessario che B sia un sottoinsieme di A

Insieme con array non ordinato

- Scriviamo innanzitutto l'interfaccia **Set**

```
public interface Set extends Container
{  void add(Object obj);
   boolean contains(Object obj);
   Object[] toArray();
}
```

- La classe **ArraySet** ha questa interfaccia pubblica

```
public class ArraySet implements Set
{  public void makeEmpty() { }
   public boolean isEmpty() { return true; }
   public void add(Object x) { }
   public boolean contains(Object x) { return true; }
   public Object[] toArray() { return null; }
}
```

  - Abbiamo scritto enunciati **return** per metodi che non restituiscono **void**
    - In questo modo la classe si compila da subito

La classe ArraySet

```
public class ArraySet implements Set
{ public ArraySet()
  { v = new Object[INITSIZE];
    vSize = 0;}
  public void makeEmpty() { vSize = 0; }
  public boolean isEmpty() { return (vSize == 0); }

  public void add(Object x)//prestazioni O(n) (usa contains)
  { if (contains(x)) return;
    if (vSize == v.length) v = resize(2*vSize);
    v[vSize++] = x; }

  public boolean contains(Object x) //metodo con prestaz. O(n)
  { for (int i = 0; i < vSize; i++)
      if (v[i].equals(x)) return true;//non si puo` usare
    return false;        //compareTo perche` x e` solo un Object

  public Object[] toArray()  // metodo con prestazioni O(n).
  { Object[] x = new Object[vSize];   //Creiamo un nuovo array
    System.arraycopy(v, 0, x, 0, vSize);//altrimenti si viola
    return x; }                          //l'incapsulamento
  private Object[] resize(int n) { ... }//solito codice
  //campi di esemplare e var. statiche
  private Object[] v;
  private int vSize;
  private static int INITSIZE = 100;
}
```

Operazioni su insiemi: unione

```
public static Set union(Set s1, Set s2)
{   Set x = new ArraySet();
    // inseriamo gli elementi del primo insieme
    Object[] v = s1.toArray();
    for (int i = 0; i < v.length; i++)
        x.add(v[i]);
    // inseriamo tutti gli elementi del
    // secondo insieme, sfruttando le
    // proprietà di add (niente duplicati...)
    v = s2.toArray();
    for (int i = 0; i < v.length; i++)
        x.add(v[i]);
    return x;
}
```

- **Prestazioni**: se **contains** è **O(n)** (e, quindi, lo è anche **add**), questa operazione è **$O(n^2)$**

Operazioni su insiemi: intersezione

```
public static Set intersection(Set s1, Set s2)
{
    Set x = new ArraySet();
    Object[] v = s1.toArray();
    for (int i = 0; i < v.length; i++)
        if (s2.contains(v[i]))
            x.add(v[i]);
            // inseriamo solo gli elementi che
            // appartengono anche al secondo
            // insieme, sfruttando le proprieta'
            // di add (niente duplicati...)
    return x;
}
```

- **Prestazioni**: se **contains** è **O(n)** l'operazione di intersezione è **$O(n^2)$**

Operazioni su insiemi: sottrazione

```
public static Set subtract(Set s1, Set s2)
{
    Set x = new ArraySet();
    Object[] v = s1.toArray();
    for (int i = 0; i < v.length; i++)
       if (!s2.contains(v[i]))
          x.add(v[i]);
          // inseriamo solo gli elementi che
          // *non* appartengono al secondo
          // insieme, sfruttando le proprieta'
          // di add (niente duplicati...)
    return x;
}
```

- **Prestazioni**: se **contains** è **O(n)** l'operazione di sottrazione è **$O(n^2)$**

Insieme con array non ordinato

- Riassumendo, realizzando un insieme con un array non ordinato
  - le prestazioni di tutte le operazioni primitive dell'insieme sono **O(n)**
  - le prestazioni di tutte le operazioni che agiscono su due insiemi sono **$O(n^2)$**
- Si può facilmente verificare che si ottengono le stesse prestazioni realizzando l'insieme con una catena (**LinkedListSet**)

Esercizio:
Insiemi di dati ordinabili

Insieme di dati ordinabili

- Cerchiamo di capire se si possono avere prestazioni migliori quando l'insieme contiene **dati ordinabili**
  - Definiamo l'interfaccia "**insieme ordinato**"

```
public interface Set extends Container
{   void add(Object obj);
    boolean contains(Object obj);
    Object[] toArray();
}
```

```
public interface SortedSet extends Set
{   void add(Comparable obj);
    Comparable[] toSortedArray();
}
```

- Realizziamo **SortedSet** usando un array ordinato
  - dovremo definire due metodi **add**, uno dei quali **impedisce** l'inserimento di dati non ordinabili

Esercizio: la classe ArraySortedSet

```
public class ArraySortedSet implements SortedSet
{   public ArraySortedSet()
    {   v = new Comparable[INITSIZE]; vSize = 0; }
    public void makeEmpty()
    {   vSize = 0; }
    public boolean isEmpty()
    {   return (vSize == 0); }
    public void add(Object x) //metodo di Set
    {   throw new IllegalArgumentException(); }
    public void add(Comparable x) // prestazioni O(n)
    { ... }     //Da completare: ordinamento per inserimento
                //E` O(n), perche' inseriamo in un array ordinato
    public boolean contains(Object x) //prestaz. O(log n)
    { ... } // da completare: usare ricerca binaria e compareTo
    public Comparable[] toSortedArray() // prestaz. O(n)
    { ... }  //da completare (v e' gia` ordinato...)
    public Object[] toArray()  //come sopra: l'array non deve
    {  return toSortedArray(); } //essere per forza disordinato
    private Comparable[] resize(int newLength) //solito metodo
    { ... } // da completare
    //campi di esemplare e variabili statiche
    private Comparable[] v;
    private int vSize;
    private static int INITSIZE = 100;
}
```

Operazioni su insiemi ordinati

- Gli algoritmi di unione, intersezione, sottrazione per insiemi generici possono essere utilizzati anche per insiemi ordinati
  - infatti, un **SortedSet** è anche un **Set**
- Qual è la complessità dell'algoritmo di unione?
  - Rimane **O(n^2)** perché il metodo **add** è rimasto **O(n)**, a causa del ri-ordinamento (con **insertionSort**) dell'array
- **Ma** sfruttiamo ciò che sappiamo delle realizzazioni di **add** e **toSortedArray** nella classe **ArraySortedSet**
  - l'array ottenuto con il metodo **toSortedArray** è **ordinato**
  - l'inserimento nell'insieme tramite **add** usa l'algoritmo di ordinamento per inserzione **in un array ordinato**

SortedSet: unione

- Per realizzare l'**unione**, osserviamo che il problema è molto simile alla **fusione di due array ordinati**
  - come abbiamo visto in **mergeSort**, questo algoritmo di fusione (che abbiamo realizzato nel metodo ausiliario **merge**) è **O(n)**
- L'unica differenza consiste nella contemporanea **eliminazione** (cioè nel non inserimento…) di eventuali **oggetti duplicati**
  - un oggetto presente in entrambi gli insiemi dovrà essere presente una sola volta nell'insieme unione

SortedSet: unione

```
public static SortedSet union(SortedSet s1,SortedSet s2)
{  SortedSet x = new ArraySortedSet();
   Comparable[] v1 = s1.toSortedArray();
   Comparable[] v2 = s2.toSortedArray();
   int i = 0, j = 0;
   while (i < v1.length && j < v2.length) // merge
      if (v1[i].compareTo(v2[j]) < 0)
         x.add(v1[i++]);
      else if (v1[i].compareTo(v2[j]) > 0)
         x.add(v2[j++]);
      else // sono uguali
      {  x.add(v1[i++]);
         j++; }
   while (i < v1.length) x.add(v1[i++]);
   while (j < v2.length) x.add(v2[j++]);
   return x;
}
```

- Quali sono le **prestazioni** di questo metodo **union**?

SortedSet: unione

- Effettuando la fusione dei due array ordinati secondo l'algoritmo visto in **MergeSort**, gli oggetti vengono via via inseriti nell'insieme unione che si va costruendo
  - Questi inserimenti avvengono con oggetti **in ordine crescente**
- Quali sono le prestazioni di **add in questo caso**?
  - L'invocazione di **contains** ha prestazioni **O(log n)** per ogni inserimento
  - L'ordinamento per inserzione in un array ordinato, usato da **add**, ha prestazioni **O(1)** per ogni inserimento!
- **In questo caso add** ha quindi prestazioni **O(log n)**
- Quindi complessivamente il metodo statico **union** ha prestazioni **O(n log n)**

SortedSet: intersezione/sottrazione

- Quali sono le prestazioni dei metodi intersection e subtract se gli oggetti **s1** ed **s2** sono di tipo **ArraySortedSet**?
  - L'invocazione **s2.contains(v[i])** ha prestazioni **O(log n)**
  - L'invocazione **x.add(v[i])** ha **in questo caso prestazioni O(log n)**. Vale infatti il ragionamento di prima:
    - L'invocazione di **contains** in **add** ha prestazioni **O(log n)** per ogni inserimento
    - L'ordinamento per inserzione in un array ordinato, usato da **add**, ha prestazioni **O(1)** per ogni inserimento!
- Quindi complessivamente i metodi statici **intersection** e **subctract** hanno prestazioni **O(n log n)**

Collaudo di Set e SortedSet

```
import java.util.Scanner;
import java.io.*;
public class SimpleSetTester
{   public static void main(String[] args) throws IOException
    {   //creazione degli insiemi: leggo dati da file e assumo
        //che il file contenga numeri interi, uno per riga
        Scanner file1 = new Scanner(new FileReader("ins1.txt"));
        Set insieme1 = new ArraySet();
        //SortedSet insieme1 = new ArraySortedSet();
        while (file1.hasNextLine())
            insieme1.add(Integer.parseInt(file1.nextLine()));
        System.out.println("\n\n*** Insieme 1 ***");
        printSet(insieme1);
        Scanner file2 = new Scanner(new FileReader("ins2.txt"));
        Set insieme2 = new ArraySet();
        //SortedSet insieme2 = new ArraySortedSet();
        while (file2.hasNextLine())
            insieme2.add(Integer.parseInt(file2.nextLine()));
        System.out.println("\n\n*** Insieme 2 ***");
        printSet(insieme2);

        file1.close();
        file2.close();                                    //continua
```

Collaudo di Set e SortedSet

```
                                                        //continua
        //Collaudo metodi di unione, intersezione, differenza

        Set unione = union(insieme1, insieme2);
        //SortedSet unione = union(insieme1, insieme2);
        System.out.println("\n\n*** Insieme Unione ***");
        printSet(unione);

        Set intersezione = intersection(insieme1, insieme2);
        System.out.println("\n\n*** Insieme Intersezione ***");
        printSet(intersezione);

        Set differenza1 = subtract(insieme1, insieme2);
        System.out.println("\n\n*** Insieme diff (1 - 2) ***");
        printSet(differenza1);

        Set differenza2 = subtract(insieme2, insieme1);
        System.out.println("\n\n*** Insieme diff (2 - 1) ***");
        printSet(differenza2);
    }
```

Collaudo di Set e SortedSet

```
    public static void printSet(Set s)
    {   Object[] array = s.toArray(); //collaudo metodo toArray
        for (int i = 0; i < array.length; i++)
            System.out.print(array[i] + " ");
        System.out.println();
    }

    public static Set union(Set s1, Set s2)
    { ... }   //codice scritto prima

    public static SortedSet union(SortedSet s1,SortedSet s2)
    { ... }   //codice scritto prima

    public static Set intersection(Set s1, Set s2)
    { ... }   //codice scritto prima

    public static Set subtract(Set s1, Set s2)
    { ... }   //codice scritto prima

}
```

Riassunto: dati in sequenza

- Abbiamo visto diversi tipi di **contenitori** per dati in sequenza, rappresentati dagli **ADT**
  - pila
  - coda
  - coda doppia
  - dizionario (mappa)
  - insieme
- Per realizzare tali **ADT**, abbiamo finora sempre usato la stessa **struttura dati**
  - array