Capitolo 15
Strutture di dati avanzate

Cay S. Horstmann
Concetti di informatica e fondamenti di Java
quarta edizione

Obiettivi del capitolo

- Apprendere i tipi di dati insieme e mappa
- Capire la realizzazione di tabelle hash
- Saper programmare funzioni di hash
- Imparare gli alberi binari
- Saper usare insiemi e mappe realizzate con alberi
- Acquisire familiarità con la struttura *heap*
- Imparare a realizzare il tipo di dati coda prioritaria
- Capire come si possano usare *heap* per effettuare ordinamenti

Insiemi

- Un insieme è una raccolta non ordinata di oggetti distinti
- Gli elementi vi possono essere aggiunti, rimossi e ricercati
- Gli insiemi non contengono duplicati. L'aggiunta del duplicato di un elemento già presente viene ignorata silenziosamente

Un insieme di stampanti

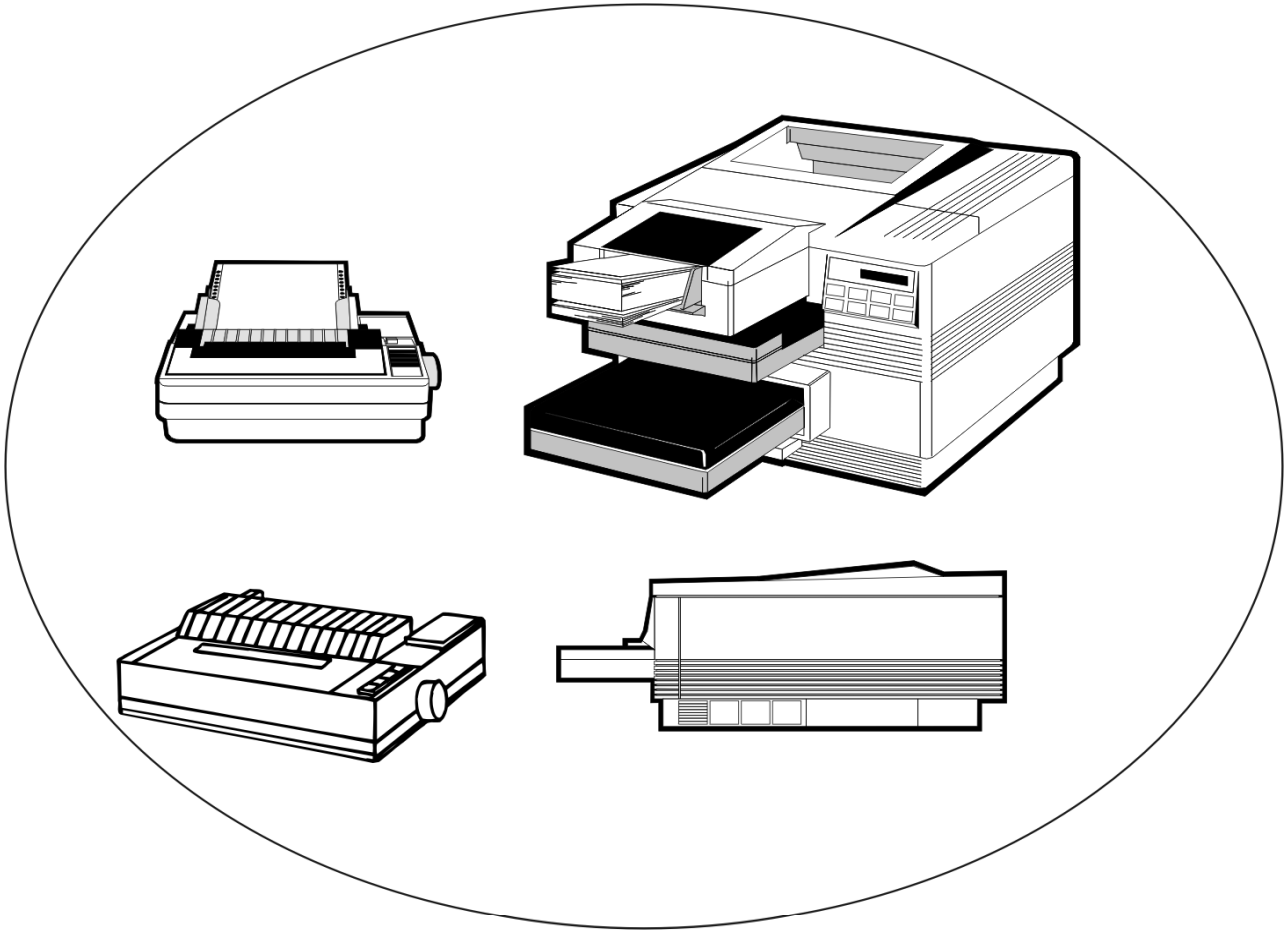


Figura 1

Operazioni fondamentali per un insieme

- Aggiunta di un elemento
- l'aggiunta del duplicato di un elemento già presente viene ignorata silenziosamente
- Rimozione di un elemento
- la rimozione di un elemento che non si trova nell'insieme viene ignorata silenziosamente
- Verifica di appartenenza (un particolare oggetto appartiene all'insieme?)
- Elenco di tutti gli elementi (in ordine arbitrario)

Insiemi

- Potremmo ovviamente usare una lista concatenata per realizzare un insieme, ma l'aggiunta, la rimozione e la verifica di appartenenza sarebbero abbastanza lente
- Esistono, infatti, due diverse strutture dati che realizzano tale obiettivo e che possono gestire queste operazioni molto più velocemente:
  - *tabelle hash*
  - *alberi*
- Tali strutture dati, dette rispettivamente `HashSet` e `TreeSet`, implementano entrambe l'interfaccia `Set`

Classi e interfacce per gli insiemi nella libreria standard

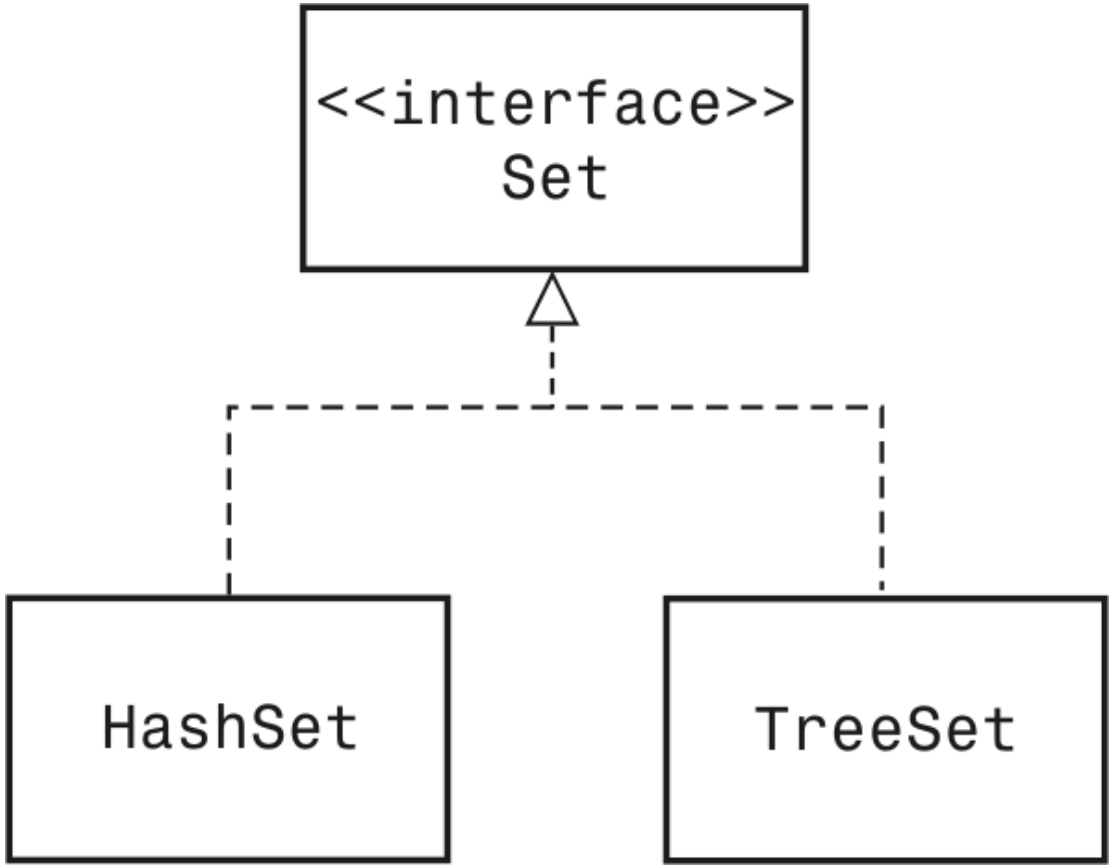


Figura 2

Iteratore

- Per elencare tutti gli elementi di un insieme si usa un iteratore
- Un iteratore dell'insieme non visita gli elementi nello stesso ordine in cui sono stati inseriti
- Il codice che realizza l'insieme conserva gli elementi in un ordine che ne consente una rapida ricerca
- Non ha senso aggiungere un elemento in una particolare posizione di un iteratore di un insieme

Codice per creare un HashSet

```
//Insieme realizzato con tabella Hash
Set<String> names = new HashSet<String>();
```

```
//Aggiunge un elemento
names.add("Romeo");
```

```
//Rimuove un elemento
names.remove("Juliet");
```

```
//Verifica se un elemento appartiene all'insieme
if (names.contains("Juliet") { . . .}
```

Elencare tutti gli elementi con un iteratore

```
Iterator<String> iter = names.iterator();

while (iter.hasNext())
   {
      String name = iter.next();
      Fai qualcosa con name
   }


// Oppure si può usare un ciclo for generalizzato
   for(String name : names)
   {
      Fai qualcosa con name
   }
```

File SetDemo.java

```
01: import java.util.HashSet;
02: import java.util.Scanner;
03: import java.util.Set;
04:
05:
06: /**
07: Programma che illustra il funzionamento di un insieme di
   stringhe
08:  L'utente può inserire e rimuovere stringhe.
09:
10: */
11: public class SetDemo
12: {
13:    public static void main(String[] args)
14:    {
15:       Set<String> names = new HashSet<String>();
16:       Scanner in = new Scanner(System.in);
17:
```

Continua…

File SetDemo.java

```
18:         boolean done = false;
19:         while (!done)
20:         {
21:            System.out.print("Add name, Q when done: ");
22:            String input = in.next();
23:            if (input.equalsIgnoreCase("Q"))
24:               done = true;
25:            else
26:            {
27:               names.add(input);
28:               print(names);
29:            }
30:         }
31:
32:         done = false;
33:         while (!done)
34:         {
```

Continua…

File SetDemo.java

```
35:             System.out.print("Remove name, Q when done");
36:             String input = in.next();
37:             if (input.equalsIgnoreCase("Q"))
38:                done = true;
39:             else
40:             {
41:                names.remove(input);
42:                print(names);
43:             }
44:          }
45:       }
46:
47:       /**
48:          Visualizza il contenuto di un insieme di stringhe.
49:          @param s un insieme di stringhe
50:       */
51:       private static void print(Set<String> s)
52:       {
```

Continua…

File SetDemo.java

```
53:         System.out.print("{ ");
54:         for (String element : s)
55:         {
56:             System.out.print(element);
57:             System.out.print(" ");
58:         }
59:         System.out.println("}");
60:     }
61: }
62:
63:
```

File SetDemo.java

- Visualizza

```
Add name, Q when done: Dick
{ Dick }
Add name, Q when done: Tom
{ Tom Dick }
Add name, Q when done: Harry
{ Harry Tom Dick }
Add name, Q when done: Tom
{ Harry Tom Dick }
Add name, Q when done: Q
Remove name, Q when done: Tom
{ Harry Dick }
Remove name, Q when done: Jerry
{ Harry Dick }
Remove name, Q when done: Q
```

Mappe

- Una mappa è un tipo di dati che memorizza associazioni tra *chiavi* (*key*) e *valori* (*value*)
- Parlando in termini matematici, una mappa è una funzione da un insieme, l'*insieme delle chiavi*, a un altro insieme, l'*insieme dei valori*
- Ogni chiave nella mappa ha un unico valore, ma un valore può essere associato a più chiavi
- La libreria Java fornisce due realizzazioni anche per le mappe: `HashMap` e `TreeMap`

Una mappa

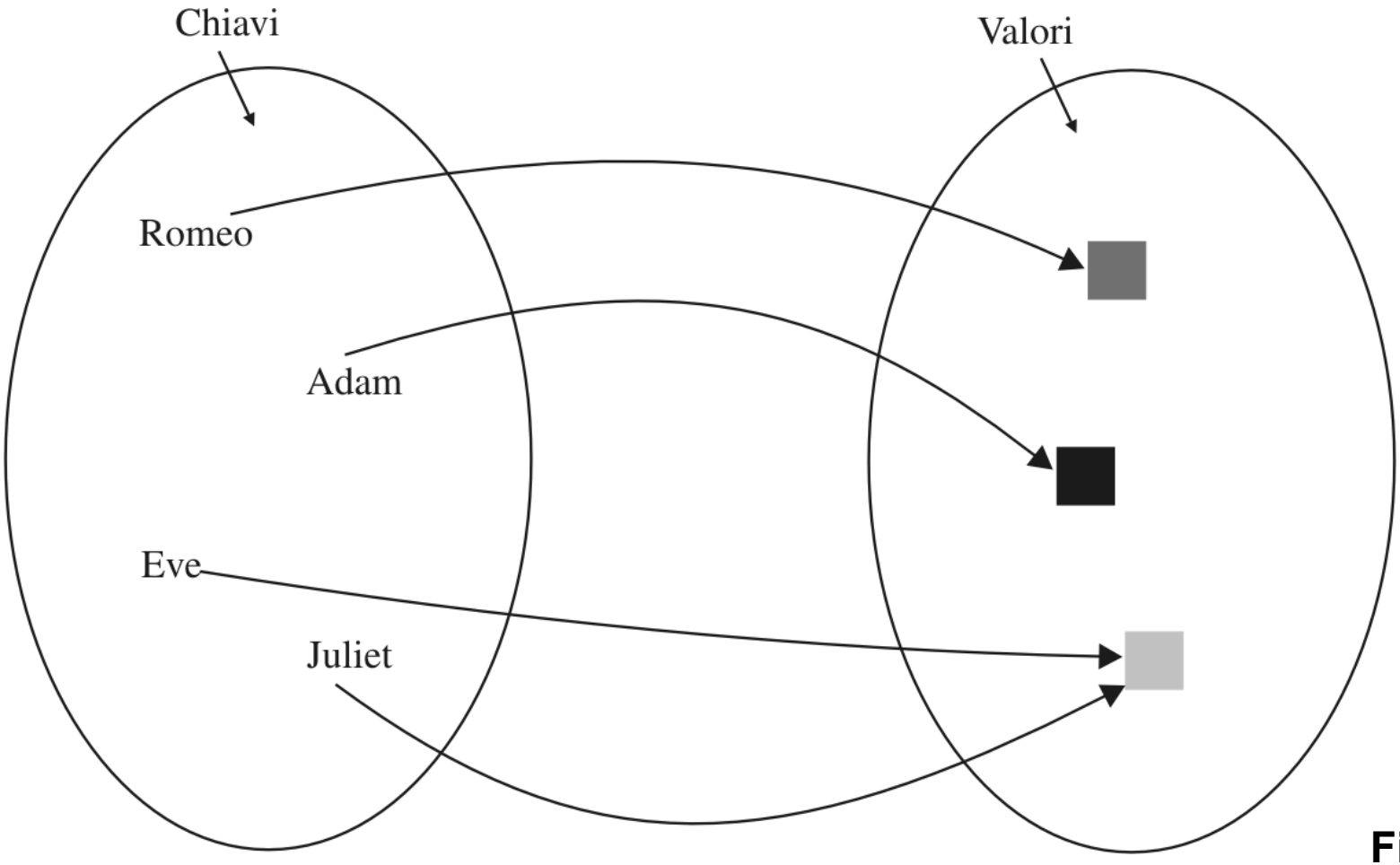


Figura 3

Classi e interfacce per le mappe nella libreria standard

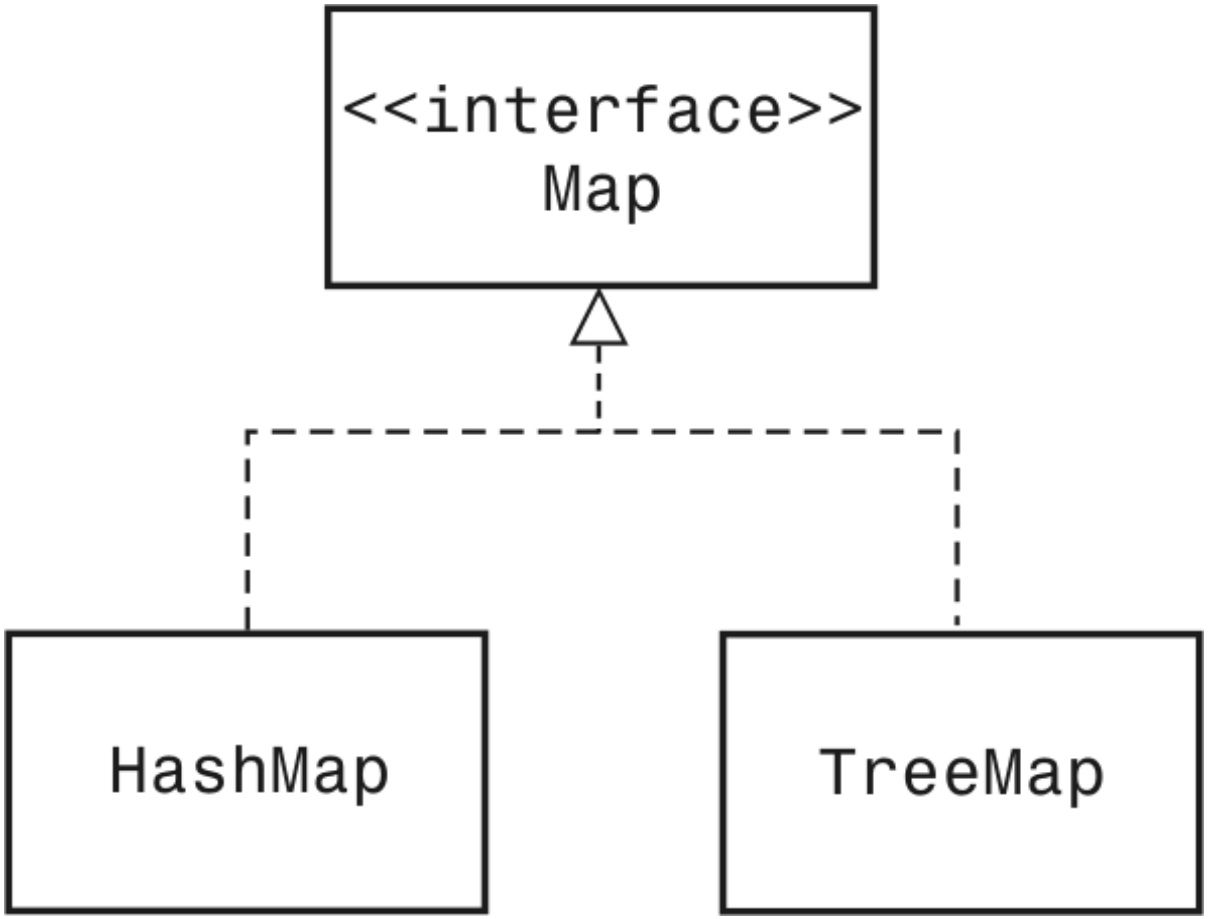


Figura 4

Creare e usare una `HashMap`

- Potete modificare il valore di un'associazione già esistente, invocando semplicemente il metodo `put` di nuovo:
  `favoriteColors.put("Juliet", Color.RED)`

- Per rimuovere una chiave e il valore a essa associato, si usa il metodo `remove`:
  `favoriteColors.remove("Juliet")`

Creare e usare una HashMap

```
//Creare una HashMap
   Map<String, Color> favoriteColors
      = new HashMap<String, Color>();
```

```
//Aggiungere un'associazione
favoriteColors.put("Juliet", Color.PINK);
```

```
//modificare un'associazione esistente
favoriteColor.put("Juliet",Color.RED);
```

Continua…

Creare e usare una HashMap

```
//restituisce il valore associato a una chiave
Color julietsFavoriteColor
      = favoriteColors.get("Juliet");
```

```
//rimuove una chiave e il valore ad essa associato
favoriteColors.remove("Juliet");
```

Coppie chiave/valore

```
Set<String> keySet = m.keySet();
for (String key : keySet)
{
   Color value = m.get(key);
   System.out.println(key + "->" + value);
}
```

File MapDemo.java

```
01: import java.awt.Color;
02: import java.util.HashMap;
03: import java.util.Map;
04: import java.util.Set;
05:
06:
07: /**
08:    Questo programma collauda una mappa che associa nomi a colori
09: */
10: public class MapDemo
11: {
12:    public static void main(String[] args)
13:    {
14:       Map<String, Color> favoriteColors
15:             = new HashMap<String, Color>();
16:       favoriteColors.put("Juliet", Color.pink);
17:       favoriteColors.put("Romeo", Color.green);
```

Continua…

File MapDemo.java

```
18:         favoriteColors.put("Adam", Color.blue);
19:         favoriteColors.put("Eve", Color.pink);
20:
21:         Set<String> keySet = favoriteColors.keySet();
22:         for (String key : keySet)
23:         {
24:            Color value = favoriteColors.get(key);
25:            System.out.println(key + "->" + value);
26:         }
27:     }
28: }
```

File MapDemo.java

- Visualizza

```
Romeo->java.awt.Color[r=0,g=255,b=0]
Eve->java.awt.Color[r=255,g=175,b=175]
Adam->java.awt.Color[r=0,g=0,b=255]
Juliet->java.awt.Color[r=255,g=175,b=175]
```

Tabelle hash

- La tecnica dell'*hashing* cerca velocemente elementi in una struttura dati, senza fare una ricerca lineare
- Una *tabella hash* può essere usata per realizzare insiemi e mappe
- Una *funzione di hash* è una funzione che, a partire da un oggetto, calcola un valore intero, il *codice di hash*
- Una buona funzione di hash minimizza le collisioni, che avvengono quando a oggetti diversi vengono associati codici di hash identici
- L'invocazione

```
int h = x.hashCode();
```

calcola il codice di hash dell'oggetto x.

Esempi di stringhe e dei relativi codici di hash

String	Hash Code
"Adam"	2035631
"Eve"	70068
"Harry"	69496448
"Jim"	74478
"Joe"	74676
"Juliet"	-2065036585
"Katherine"	2079199209
"Sue"	83491

Una realizzazione molto semplice di tabella hash

- Realizzazione:
- Creare un codice di hash
- Costruire un array
- Inserire ciascun oggetto nella posizione indicata dal suo codice di hash
- Scoprire se un oggetto sia già presente nell'insieme oppure no:
- Calcolare il suo codice di hash
- Verificare se la posizione dell'array con quel codice è già occupata

Una realizzazione molto semplice di tabella hash

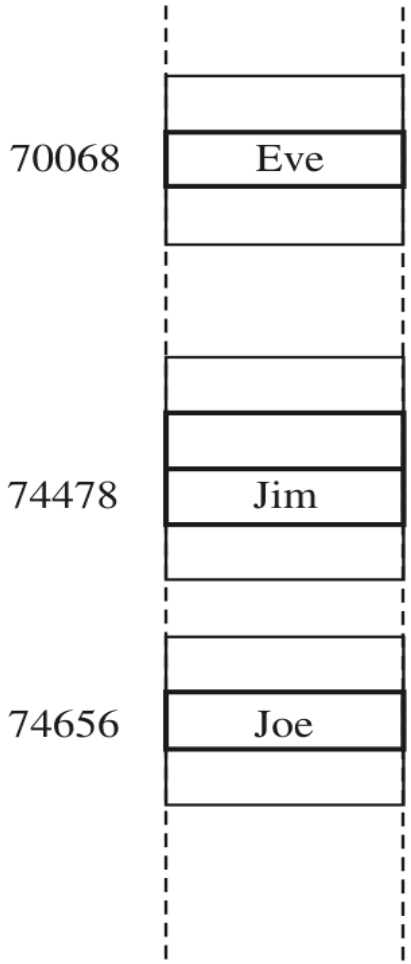


Figura 5

Problemi di una realizzazione molto semplice di tabella hash

- Non è possibile creare un array che sia tanto grande da contenere tutte le posizioni indicate dai numeri interi possibili
- E' possibile che due oggetti diversi abbiano lo stesso codice di hash

Soluzioni

- Scegliere un array con una qualche ragionevole dimensione e costringere il codice di hash a ricadere in tale intervallo

```
int h = x.hashCode();
if (h < 0) h = -h;
h = h % size;
```

- Per memorizzare più oggetti nella stessa posizione dell'array, usiamo brevi sequenze concatenate per gli elementi che hanno lo stesso codice di hash. Queste sequenze concatenate vengono chiamate *bucket*.

Una tabella hash con *bucket* per memorizzare elementi aventi lo stesso codice di hash

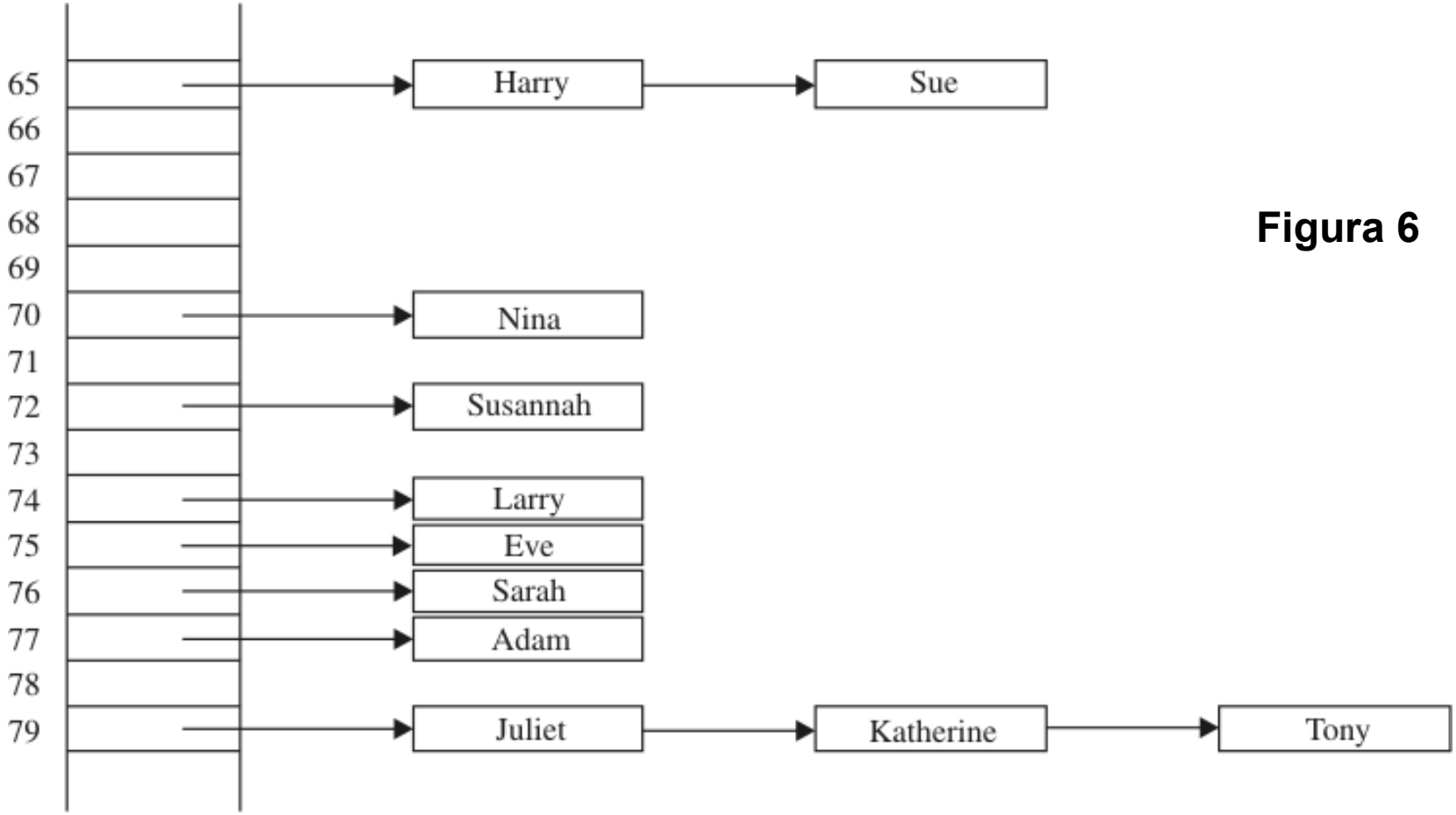


Figura 6

Algoritmo per trovare un oggetto x in una tabella hash

- Si calcola il codice di hash e lo si riduce modulo la dimensione della tabella, generando un indice `h` all'interno della tabella hash.
- Si itera attraverso gli elementi del bucket che si trova in posizione `h`. Per ogni elemento del bucket, si verifica se è uguale a `x`.
- Se si trova una corrispondenza fra gli elementi di tale bucket, allora `x` appartiene all'insieme. Altrimenti no.

Tabelle hash

- Una tabella hash può essere realizzata come array di bucket, che sono sequenze di nodi che contengono elementi aventi lo stesso codice di hash
- Se non ci sono collisioni o se ce ne sono poche, aggiungere, eliminare e cercare elementi nella tabella hash richiede un tempo costante, O(1)
- La dimensione della tabella dovrebbe essere un numero primo, più grande del numero di elementi previsto
- Una capacità eccedente del 30% è un valore tipico

Tabelle hash

- L'inserimento di un elemento è una semplice estensione dell'algoritmo utilizzato per trovare un oggetto
- Si calcola il codice di hash per trovare il bucket in cui l'elemento dovrebbe essere inserito
- Si cerca l'elemento in quel bucket
- Se è già presente, non si fa nulla, altrimenti lo si inserisce.

Tabelle hash

- Eliminare un elemento è altrettanto semplice
- Si calcola il codice di hash per trovare il bucket in cui l'elemento dovrebbe essere inserito
- Si cerca l'elemento in quel bucket
- Se è presente, lo si rimuove, altrimenti non si fa nulla
- Finché ci sono poche collisioni, un elemento può essere aggiunto o rimosso in un tempo costante, $O(1)$

File HashSet.java

```
001: import java.util.AbstractSet;
002: import java.util.Iterator;
003: import java.util.NoSuchElementException;
004:
005: /**
006:    Un esemplare di HashSet memorizza una raccolta
007:    non ordinata di oggetti usando una tabella hash.
008: */
009: public class HashSet extends AbstractSet
010: {
011:    /**
012:       Costruisce una tabella hash.
013:       @param bucketsLength la lunghezza dell'array di bucket
014:    */
015:    public HashSet(int bucketsLength)
016:    {
```

Continua…

File HashSet.java

```
017:        buckets = new Node[bucketsLength];
018:        size = 0;
019:     }
020:
021:     /**
022:        Verifica l'appartenenza all'insieme.
023:        @param x un oggetto
024:        @return true se x è un elemento dell'insieme
025:     */
026:     public boolean contains(Object x)
027:     {
028:        int h = x.hashCode();
029:        if (h < 0) h = -h;
030:        h = h % buckets.length;
031:
032:        Node current = buckets[h];
033:        while (current != null)
034:        {
```

Continua…

File HashSet.java

```
035:             if (current.data.equals(x)) return true;
036:             current = current.next;
037:          }
038:          return false;
039:       }
040:
041:       /**
042:          Aggiunge un elemento all'insieme.
043:          @param x un oggetto
044:          @return true se x è un oggetto nuovo,
                     false se x era già presente nell'insieme
046:       */
047:       public boolean add(Object x)
048:       {
049:          int h = x.hashCode();
050:          if (h < 0) h = -h;
051:          h = h % buckets.length;
052:
```

File HashSet.java

```
053:         Node current = buckets[h];
054:         while (current != null)
055:         {
056:            if (current.data.equals(x))
057:               return false; // Già presente nell'insieme
058:            current = current.next;
059:         }
060:         Node newNode = new Node();
061:         newNode.data = x;
062:         newNode.next = buckets[h];
063:         buckets[h] = newNode;
064:         size++;
065:         return true;
066:      }
067:
```

File HashSet.java

```
068:    /**
069:       Elimina un oggetto dall'insieme.
070:       @param x un oggetto
071:       @return true se x viene eliminato dall'insieme,
                   false se x non è presente nell'insieme
073:    */
074:    public boolean remove(Object x)
075:    {
076:       int h = x.hashCode();
077:       if (h < 0) h = -h;
078:       h = h % buckets.length;
079:
080:       Node current = buckets[h];
081:       Node previous = null;
082:       while (current != null)
083:       {
084:          if (current.data.equals(x))
085:          {
```

File HashSet.java

```
086:                if (previous == null) buckets[h] = current.next;
087:                else previous.next = current.next;
088:                size--;
089:                return true;
090:             }
091:             previous = current;
092:             current = current.next;
093:          }
094:          return false;
095:       }
096:
097:       /**
098:          Restituisce un iteratore che attraversa gli elementi
           dell'insieme.
099:          @param a hash set iterator
100:       */
101:       public Iterator iterator()
102:       {
103:          return new HashSetIterator();
104:       }
```

File HashSet.java

```
105:
106:    /**
107:       Restituisce il numero di elementi nell'insieme.
108:       @return il numero di elementi
109:    */
110:    public int size()
111:    {
112:       return size;
113:    }
114:
115:    private Node[] buckets;
116:    private int size;
117:
118:    private class Node
119:    {
120:       public Object data;
121:       public Node next;
122:    }
123:
```

File HashSet.java

```
124:    private class HashSetIterator implements Iterator
125:    {
126:       /**
127:       Costruisce un iteratore per insiemi hash che punta
128:          al primo elemento dell'insieme.
129:       */
130:       public HashSetIterator()
131:       {
132:          current = null;
133:          bucket = -1;
134:          previous = null;
135:          previousBucket = -1;
136:       }
137:
138:       public boolean hasNext()
139:       {
140:          if (current != null && current.next != null)
141:             return true;
```

File HashSet.java

```
142:             for (int b = bucket + 1; b < buckets.length; b++)
143:                if (buckets[b] != null) return true;
144:             return false;
145:          }
146:
147:          public Object next()
148:          {
149:             previous = current;
150:             previousBucket = bucket;
151:             if (current == null || current.next == null)
152:             {
153:                // avanza al prossimo bucket
154:                bucket++;
155:
156:                while (bucket < buckets.length
157:                      && buckets[bucket] == null)
158:                   bucket++;
```

File HashSet.java

```
159:                if (bucket < buckets.length)
160:                   current = buckets[bucket];
161:                else
162:                   throw new NoSuchElementException();
163:             }
164:             else // avanza al prossimo elemento nel bucket
165:                current = current.next;
166:             return current.data;
167:          }
168:
169:          public void remove()
170:          {
171:             if (previous != null && previous.next == current)
172:                previous.next = current.next;
173:             else if (previousBucket < bucket)
174:                buckets[bucket] = current.next;
175:             else
176:                throw new IllegalStateException();
```

File HashSet.java

```
177:             current = previous;
178:             bucket = previousBucket;
179:          }
180:
181:          private int bucket;
182:          private Node current;
183:          private int previousBucket;
184:          private Node previous;
185:       }
186: }
```

File SetDemo.java

```
01: import java.util.Iterator;
02: import java.util.Set;
03:
04: /**
05:    Questo programma collauda la classe insieme realizzata
       con tabelle hash
06: */
07: public class SetDemo
08: {
09:    public static void main(String[] args)
10:    {
11:       HashSet names = new HashSet(101); // 101 è primo
12:
13:       names.add("Sue");
14:       names.add("Harry");
15:       names.add("Nina");
16:       names.add("Susannah");
17:       names.add("Larry");
18:       names.add("Eve");
```

Continua…

File SetDemo.java

```
19:         names.add("Sarah");
20:         names.add("Adam");
21:         names.add("Tony");
22:         names.add("Katherine");
23:         names.add("Juliet");
24:         names.add("Romeo");
25:         names.remove("Romeo");
26:         names.remove("George");
27:
28:         Iterator iter = names.iterator();
29:         while (iter.hasNext())
30:            System.out.println(iter.next());
31:     }
32: }
```

File `SetDemo.java`

- Visualizza

```
Harry
Sue
Nina
Susannah
Larry
Eve
Sarah
Adam
Juliet
Katherine
Tony
```

Calcolare codici hash

- Una funzione di hash calcola un codice di hash intero a partire da un oggetto in modo che sia probabile che oggetti diversi abbiano un codice diverso
- Bisogna combinare i valori dei caratteri della stringa per ottenere un numero intero; si potrebbe, ad esempio, sommare i valori dei caratteri

```
int h = 0;
for (int i = 0; i < s.length(); i++)
   h = h + s.charAt(i);
```

- Questa, però, non sarebbe una buona idea, perché non mescola sufficientemente i valori dei caratteri. Stringhe che sono permutazioni di un'altra (come `eat` e `tea`) avrebbero lo stesso codice hash.

Calcolare codici hash

- Ecco invece il metodo usato nella libreria standard per calcolare il codice di hash di una stringa

```
final int HASH_MULTIPLIER = 31;
int h = 0;
for (int i = 0; i < s.length(); i++)
   h = HASH_MULTIPLIER * h + s.charAt(i)
```

- Il codice di hash di "`eat`" è

```
31 * (31 * 'e' + 'a') + 't' = 100184
```

- Il codice di hash di "`tea`" è abbastanza diverso

```
31 * (31 * 't' + 'e') + 'a' = 114704
```

Il metodo `hashCode` per la classe `Coin`

- Il metodo `hashCode` per la classe `Coin` ha due campi di esemplare: il nome della moneta e il suo valore
- Calcoliamo il loro codice di hash
- Per calcolare il codice di hash di un numero in virgola mobile
- Costruite dapprima un oggetto di tipo `Double` che lo contenga
- Calcolate il suo codice di hash
- Combinate quindi i due codici di hash
- Usate un numero primo come moltiplicatore di hash

Il metodo hashCode per la classe Coin

```
class Coin
 {
    public int hashCode()
    {
       int h1 = name.hashCode();
       int h2 = new Double(value).hashCode();
       final int HASH_MULTIPLIER = 29;
       int h = HASH_MULTIPLIER * h1 + h2:

       return h
    }
    . . .
 }
```

Calcolare codici hash

- Usate un numero primo come moltiplicatore di hash
- Se avete più di due campi di esemplare, combinate i loro codici di hash come segue:

```
int h = HASH_MULTIPLIER * h1 +h2;
h = HASH_MULTIPLIER * h + h3;
h = HASH_MULTIPLIER *h + h4;
  . . .
return h;
```

Calcolare codici hash

- Il metodo `hashCode` deve essere compatibile con il metodo `equals`

Se `x.equals(y)`, allora `x.hashCode() == y.hashCode()`

Calcolare codici hash

- Se la vostra classe definisce il metodo `equals` ma non il metodo `hashCode` ci saranno dei problemi
- Supponiamo di esserci dimenticati di definire il metodo `hashCode` per la classe `Coin`: di conseguenza, essa eredita il metodo per il calcolo del codice di hash dalla superclasse `Object`, che calcola un codice di hash a partire dall'*indirizzo in memoria* dell'oggetto.
- L'effetto è una probabilità molto alta che due qualsiasi oggetti abbiano codici di hash diversi

```
Coin coin1 = new Coin(0.25, "quarter");
Coin coin2 = new Coin(0.25, "quarter");
```

Mappe hash

- Quando usate un oggetto di tipo `HashMap`, vengono calcolati i codici di hash soltanto per le chiavi
- Le chiavi devono avere metodi `equals` e `hashCode` compatibili

File Coin.java

```
01: /**
02:    Una moneta con un valore.
03: */
04: public class Coin
05: {
06:    /**
07:       Costruisce una moneta.
08:       @param aValue il valore della moneta
09:       @param aName il nome della moneta
10:    */
11:    public Coin(double aValue, String aName)
12:    {
13:       value = aValue;
14:       name = aName;
15:    }
16:
```

Continua…

File Coin.java

```
17:    /**
18:       Restituisce il valore della moneta.
19:       @return il valore
20:    */
21:    public double getValue()
22:    {
23:       return value;
24:    }
25:
26:    /**
27:       Restituisce il nome della moneta.
28:       @return il nome
29:    */
30:    public String getName()
31:    {
32:       return name;
33:    }
34:
```

Continua…

File Coin.java

```
35:    public boolean equals(Object otherObject)
36:    {
37:       if (otherObject == null) return false;
38:       if (getClass() != otherObject.getClass()) return false;
39:       Coin other = (Coin) otherObject;
40:       return value == other.value && name.equals(other.name);
41:    }
42:
43:    public int hashCode()
44:    {
45:       int h1 = name.hashCode();
46:       int h2 = new Double(value).hashCode();
47:       final int HASH_MULTIPLIER = 29;
48:       int h = HASH_MULTIPLIER * h1 + h2;
49:       return h;
50:    }
51:
```

Continua…

File Coin.java

```
52:     public String toString()
53:     {
54:        return "Coin[value=" + value + ",name=" + name + "]";
55:     }
56:
57:     private double value;
58:     private String name;
59: }
```

File CoinHashCodePrinter.java

```
01: import java.util.HashSet;
02: import java.util.Set;
03:
04:
05: /**
06:    Un programma per visualizzare i codici di hash delle monete.
07: */
08: public class CoinHashCodePrinter
09: {
10:    public static void main(String[] args)
11:    {
12:       Coin coin1 = new Coin(0.25, "quarter");
13:       Coin coin2 = new Coin(0.25, "quarter");
14:       Coin coin3 = new Coin(0.05, "nickel");
15:
```

Continua…

File CoinHashCodePrinter.java

```
16:         System.out.println("hash code of coin1="
17:               + coin1.hashCode());
18:         System.out.println("hash code of coin2="
19:               + coin2.hashCode());
20:         System.out.println("hash code of coin3="
21:               + coin3.hashCode());
22:
23:         Set<Coin> coins = new HashSet<Coin>();
24:         coins.add(coin1);
25:         coins.add(coin2);
26:         coins.add(coin3);
27:
28:         for (Coin c : coins)
29:            System.out.println(c);
30:     }
31: }
```

File `CoinHashCodePrinter.java`

- Visualizza

```
hash code of coin1=-1513525892
hash code of coin2=-1513525892
hash code of coin3=-1768365211
Coin[value=0.25,name=quarter]
Coin[value=0.05,name=nickel]
```

Alberi di ricerca binari

- Gli alberi di ricerca binari consentono rapidi inserimenti ed eliminazioni di elementi
- Sono stati concepiti appositamente per ricerche veloci
- Un albero binario è composto di nodi, ciascuno dei quali ha al massimo due nodi figli
- Tutti i nodi di un albero di ricerca binario soddisfano la seguente proprietà:
- I discendenti di sinistra contengono dati di valore inferiore al dato contenuto nel nodo
- I discendenti di destra contengono dati di valore maggiore

Un albero di ricerca binario

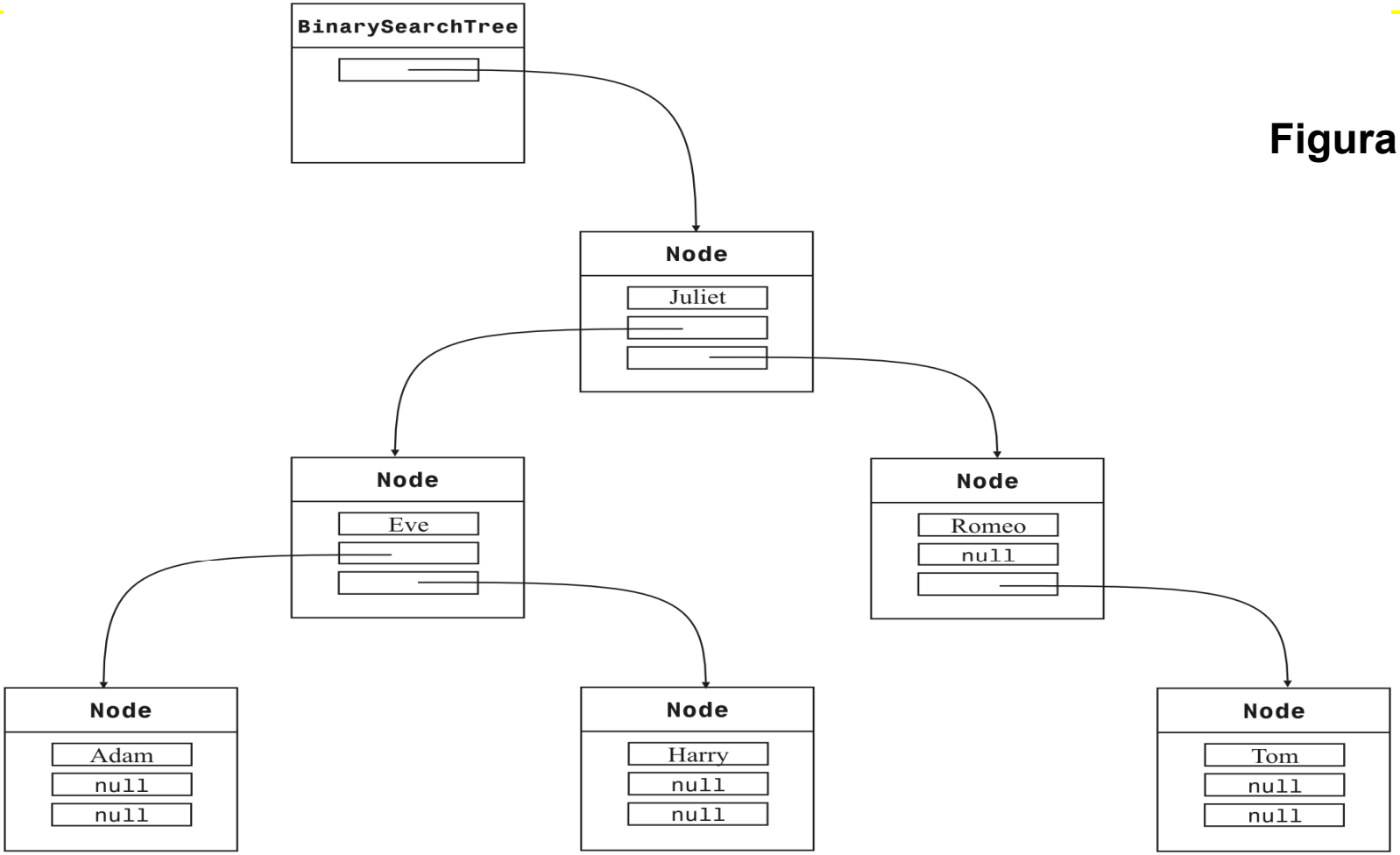


Figura 7

Un albero binario che non è un albero di ricerca binario

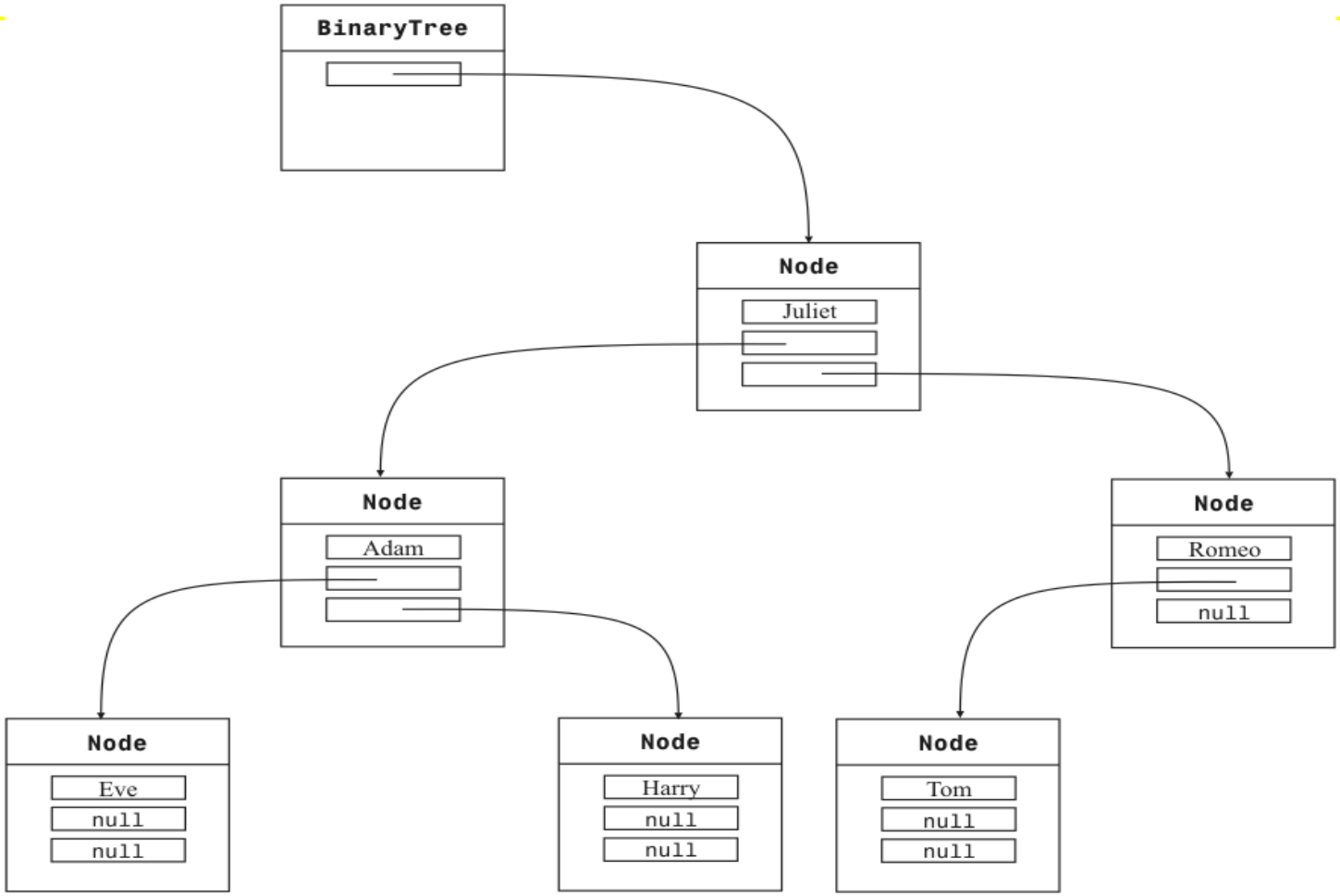


Figura 8

Realizzare un albero di ricerca binario

- Occorre una classe per l'albero, che contenga un riferimento al *nodo radice* ("root"), e una classe separata per i nodi
- Ciascun nodo contiene due riferimenti (al nodo figlio di sinistra e a quello di destra)
- Ciascun nodo contiene un campo dati
- La variabile `data` è di tipo `Comparable`, non `Object`, perché in un albero di ricerca binario dovete essere in grado di confrontare i valori per poterli collocare nelle posizioni corrette

Realizzare un albero di ricerca binario

```
public class BinarySearchTree
{
   public BinarySearchTree() {. . .}
   public void add(Comparable obj) {. . .}
   . . .
   private Node root;

   private class Node
   {
      public void addNode(Node newNode) { . . . }
      . . .
      public Comparable data;
      public Node left;
      public Node right;
   }
}
```

Algoritmo

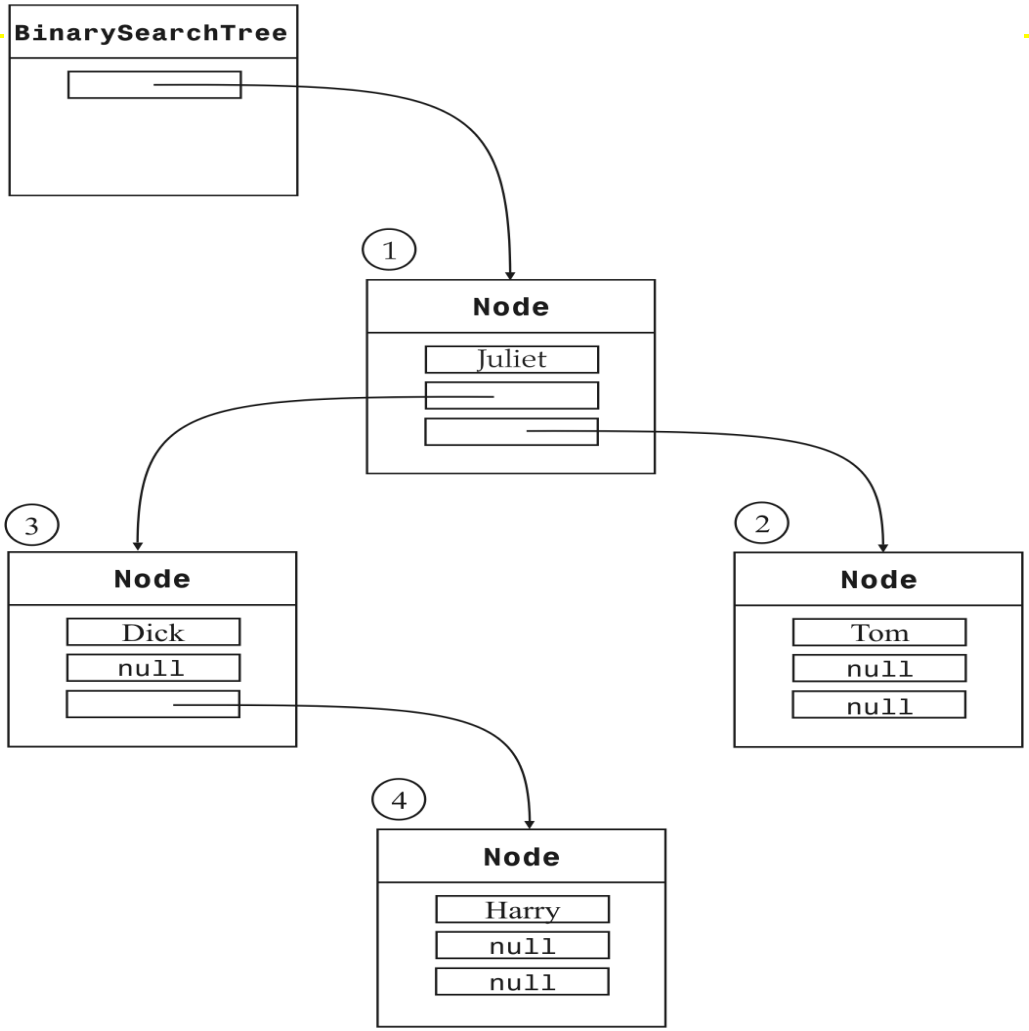
- Se trovate un puntatore a un nodo che sia diverso da `null`, esaminate il suo campo `data`
- Se il valore di `data` di quel nodo è maggiore del dato che volete inserire, continuate il processo nel sottoalbero sinistro
- Se il valore di `data` esistente è minore, continuate il processo nel sottoalbero destro
- Se trovate un puntatore `null` a un nodo, sostituitelo con il nuovo nodo

Esempio

```
BinarySearchTree tree = new BinarySearchTree();

tree.add("Juliet"); //1

tree.add("Tom"); //2

tree.add("Dick"); //3

tree.add("Harry"); //4
```

Un albero di ricerca binario dopo quattro inserimenti

Figura 9



Un albero di ricerca binario dopo cinque inserimenti

Vogliamo inserirvi un nuovo elemento:

```
tree.add("Romeo"); //5
```

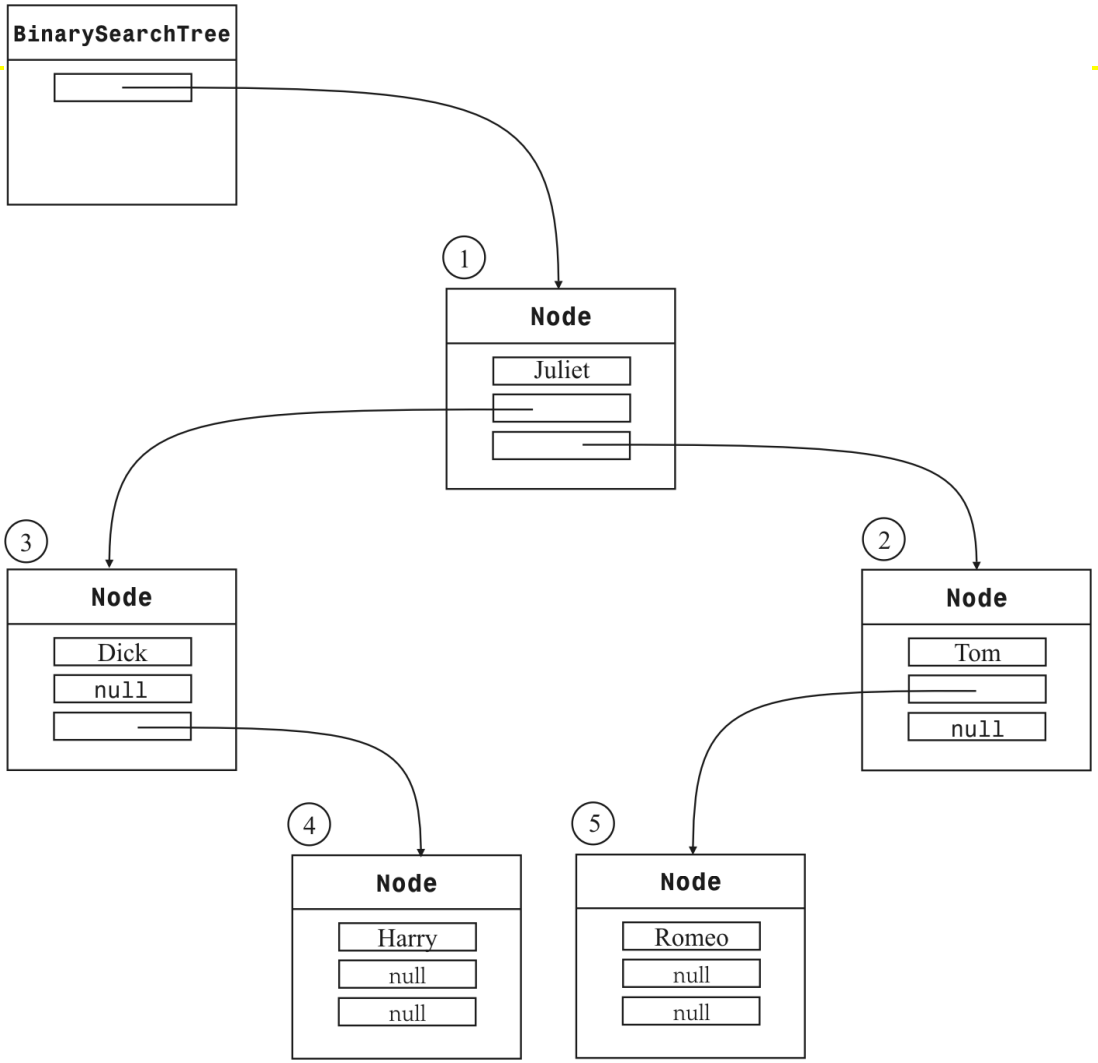


Figura 10

Codice per il metodo `add` della classe `BinarySearchTree`

```
public class BinarySearchTree
{
   . . .
   public void add(Comparable obj)
   {
      Node newNode = new Node();
      newNode.data = obj;
      newNode.left = null;
      newNode.right = null;
      if (root == null) root = newNode;
      else root.addNode(newNode);
   }
   . . .
}
```

Algoritmo della classe Node

```
private class Node
{
   . . .
   public void addNode(Node newNode)
   {
      int comp = newNode.data.compareTo(data);
      if (comp < 0)
      {
         if (left == null) left = newNode;
         else left.addNode(newNode);
      }
      else if (comp > 0)
      {
         if (right == null) right = newNode;
         else right.addNode(newNode);
      }
   }
   . . .
}
```

Albero di ricerca binario

- Eliminando da un albero di ricerca binario un nodo avente un solo figlio, tale figlio prende il posto del nodo eliminato
- Per eliminare da un albero di ricerca binario un nodo avente due figli, lo si sostituisce con il valore minimo presente nel suo sottoalbero destro

Eliminazione di un nodo avente un solo figlio

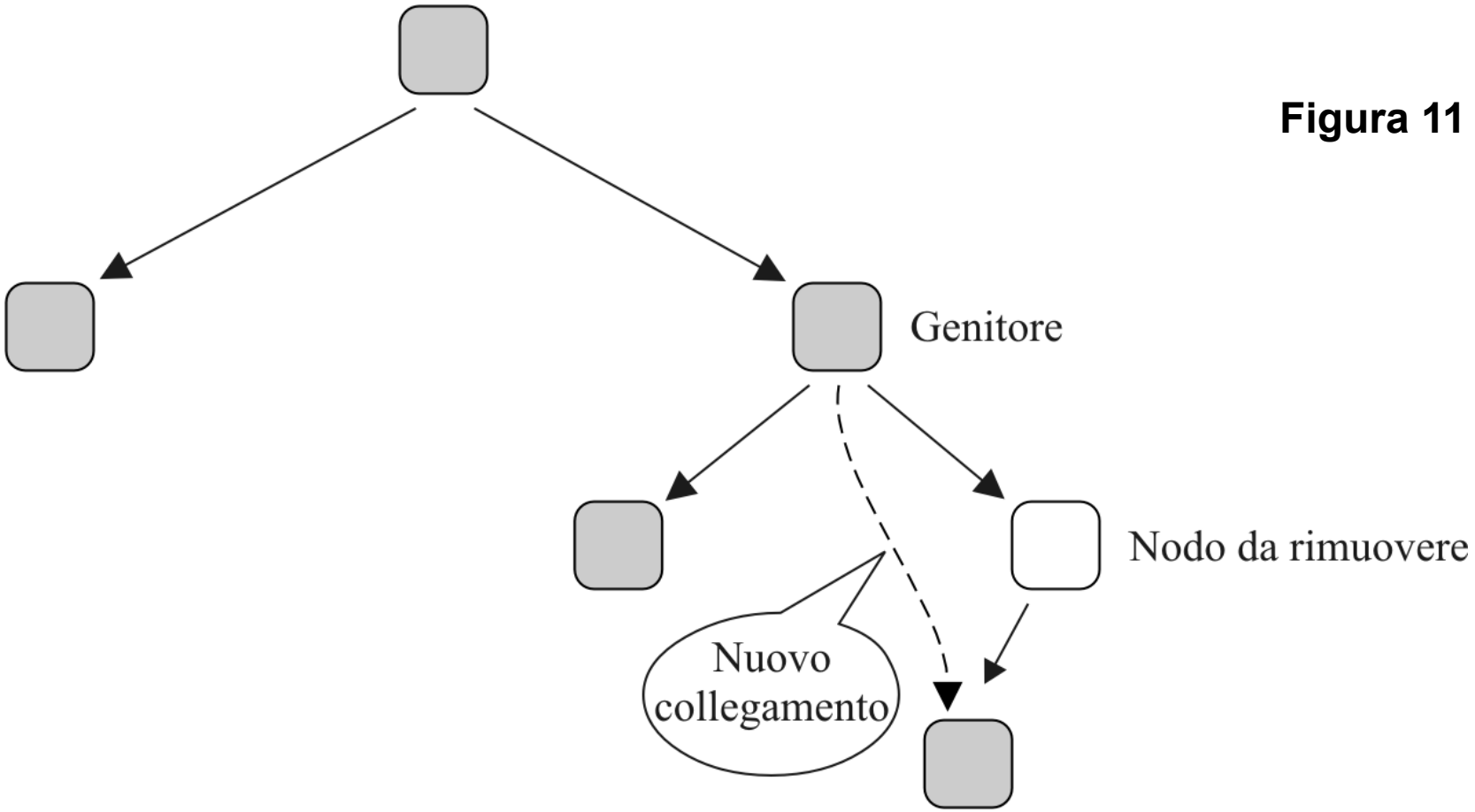


Figura 11

Eliminazione di un nodo avente due figli

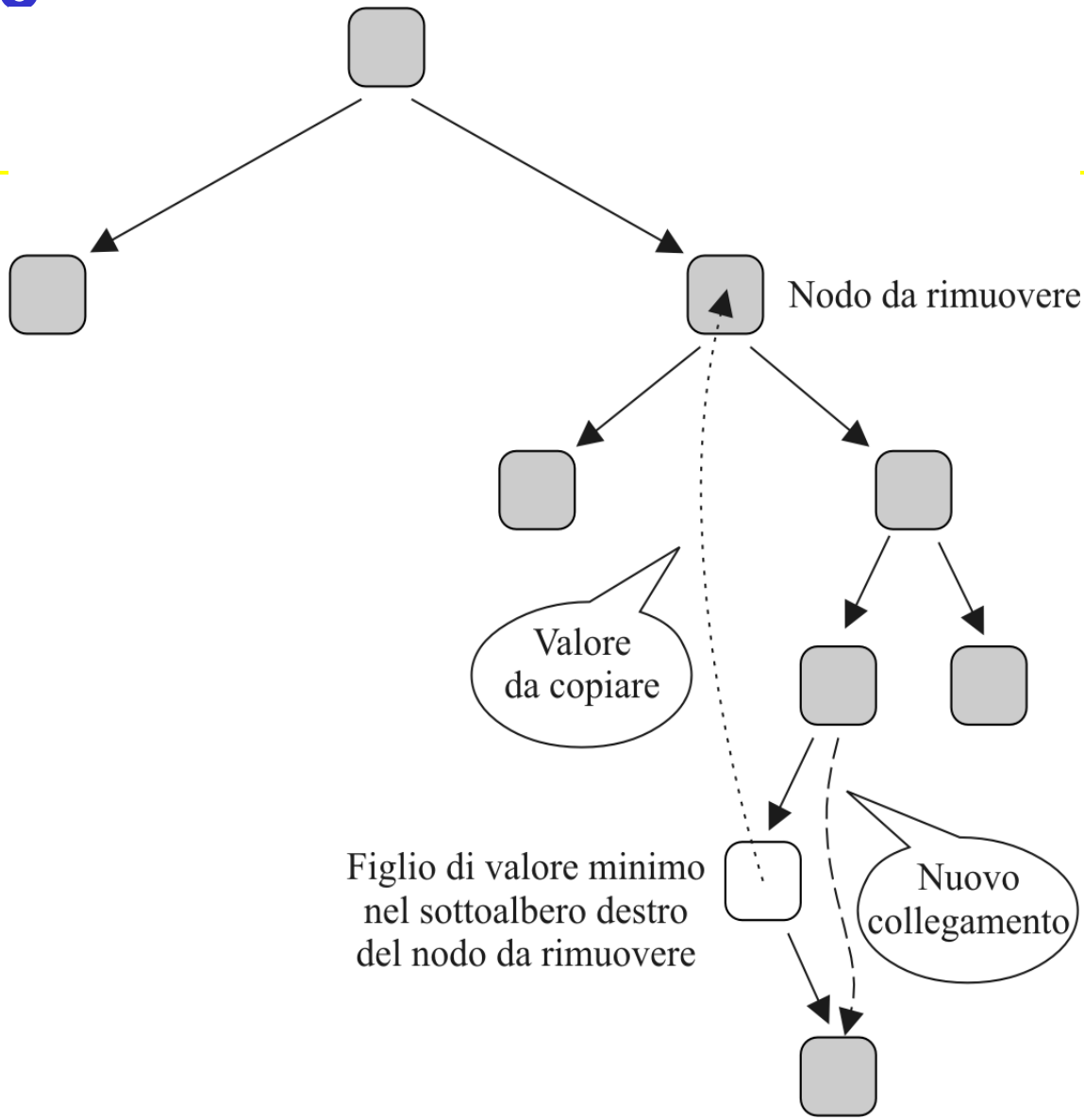


Figura 12

Alberi di ricerca binari

- Albero *bilanciato*: ogni nodo ha approssimativamente tanti discendenti a sinistra quanti ne ha a destra
- Se un albero di ricerca binario è bilanciato, aggiungervi un elemento richiede un tempo O(log n)
- Se l'albero fosse *sbilanciato*, allora l'inserimento potrebbe essere lento: non più veloce dell'inserimento in una lista concatenata

Albero di ricerca binario sbilanciato

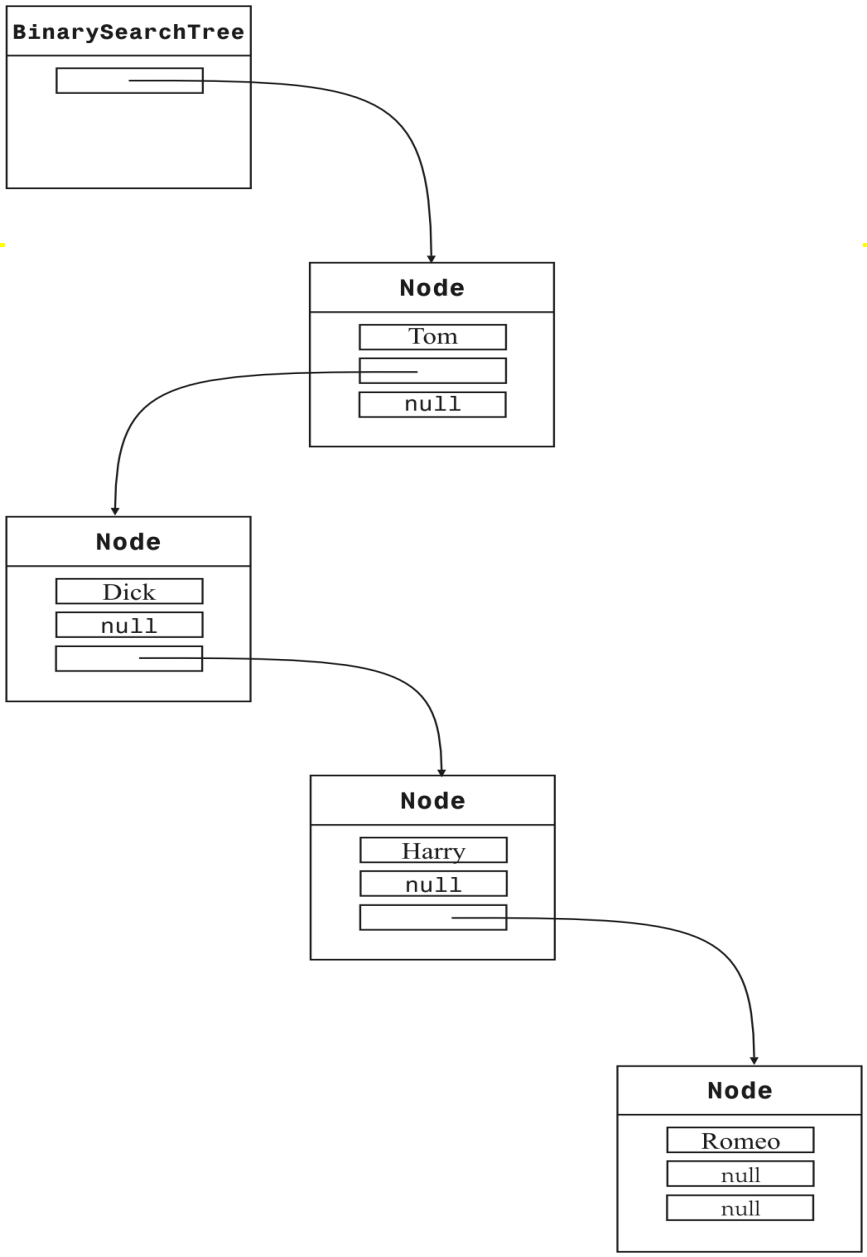


Figura 13

File BinarySearchTree.java

```
001: /**
002:    Questa classe implementa un albero di ricerca binario
003:    i cui nodi contengono oggetti che implementano
004:    l'interfaccia Comparable
005: */
006: public class BinarySearchTree
007: {
008:    /**
009:       Costruisce un albero vuoto.
010:    */
011:    public BinarySearchTree()
012:    {
013:       root = null;
014:    }
015:
```

Continua…

File BinarySearchTree.java

```
016:    /**
017:       Inserisce un nuovo nodo nell'albero.
018:       @param obj l'oggetto da inserire
019:    */
020:    public void add(Comparable obj)
021:    {
022:       Node newNode = new Node();
023:       newNode.data = obj;
024:       newNode.left = null;
025:       newNode.right = null;
026:       if (root == null) root = newNode;
027:       else root.addNode(newNode);
028:    }
029:
```

Continua…

File BinarySearchTree.java

```
030:    /**
031:       Cerca un oggetto nell'albero.
032:       @param obj l'oggetto da inserire
033:@return true se e solo se l'oggetto è presente nell'albero
034:    */
035:    public boolean find(Comparable obj)
036:    {
037:       Node current = root;
038:       while (current != null)
039:       {
040:          int d = current.data.compareTo(obj);
041:          if (d == 0) return true;
042:          else if (d > 0) current = current.left;
043:          else current = current.right;
044:       }
045:       return false;
046:    }
047:
```

Continua…

File BinarySearchTree.java

```
048:    /**
049:       Cerca di eliminare un oggetto dall'albero.
050:       Non fa nulla se l'oggetto non è presente.
051:       @param obj l'oggetto da eliminare
052:    */
053:    public void remove(Comparable obj)
054:    {
055:       // cerca il nodo da eliminare
056:
057:       Node toBeRemoved = root;
058:       Node parent = null;
059:       boolean found = false;
060:       while (!found && toBeRemoved != null)
061:       {
062:          int d = toBeRemoved.data.compareTo(obj);
063:          if (d == 0) found = true;
064:          else
065:          {
```

Continua...

File BinarySearchTree.java

```
066:                parent = toBeRemoved;
067:                if (d > 0) toBeRemoved = toBeRemoved.left;
068:                else toBeRemoved = toBeRemoved.right;
069:             }
070:          }
071:
072:          if (!found) return;
073:
074:          // toBeRemoved contiene obj
075:
076:          // se uno dei figli è vuoto, si usa l'altro
077:
078:          if (toBeRemoved.left == null
                 || toBeRemoved.right == null)
079:          {
080:             Node newChild;
081:             if (toBeRemoved.left == null)
082:                newChild = toBeRemoved.right;
```

Continua…

File BinarySearchTree.java

```
083:            else
084:               newChild = toBeRemoved.left;
085:
086:            if (parent == null) // Trovato in root
087:               root = newChild;
088:            else if (parent.left == toBeRemoved)
089:               parent.left = newChild;
090:            else
091:               parent.right = newChild;
092:            return;
093:         }
094:
095:         // nessun sottoalbero è vuoto
096:
097:         // cerca l'elemento minimo nel sottoalbero destro
098:
```

Continua…

File BinarySearchTree.java

```
099:         Node smallestParent = toBeRemoved;
100:         Node smallest = toBeRemoved.right;
101:         while (smallest.left != null)
102:         {
103:            smallestParent = smallest;
104:            smallest = smallest.left;
105:         }
106:
107:         // smallest è il nodo che contiene l'elemento minimo
             // presente nel sottoalbero destro
108:         // sposta i dati, scollega il figlio
109:
110:
111:          toBeRemoved.data = smallest.data;
              if (smallestParent = = toBeRemoved)
112:          smallestParent.right = smallest.right;
113:          else
114:           smallestParent.left = smallest.right
       }
```

Continua…

File BinarySearchTree.java

```
115:    /**
116:  Stampa il contenuto dell'albero in successione ordinata.
117:    */
118:    public void print()
119:    {
120:       if (root != null)
121:          root.printNodes();
122:       System.out.println();
123:    }
124:    private Node root;
125:
126:    /**
127:       Un nodo di un albero memorizza un dato e i
128:  riferimenti a due nodi, figlio sinistro e figlio destro.
129:    */
130:    private class Node
131:    {
```

Continua…

File BinarySearchTree.java

```
132:       /**
133:    Inserisce un nuovo nodo come discendente di questo nodo.
134:          @param newNode il nodo da inserire
135:       */
136:       public void addNode(Node newNode)
137:       {
138:          int comp = newNode.data.compareTo(data);
              if (comp < 0)
139:          {
140:             if (left == null) left = newNode;
141:             else left.addNode(newNode);
142:          }
143:          else if (comp > 0)
144:          {
145:             if (right == null) right = newNode;
146:             else right.addNode(newNode);
147:          }
148:       }
149:
```

Continua…

File BinarySearchTree.java

```
150:         /**
151:            Stampa questo nodo e tutti i suoi discendenti
152:            in successione ordinata.
153:         */
154:         public void printNodes()
155:         {
156:            if (left != null)
157:               left.printNodes();
158:            System.out.print(data + " ");
159:            if (right != null)
160:               right.printNodes();
161:         }
162:
163:         public Comparable data;
164:         public Node left;
165:         public Node right;
166:      }
167: }
```

Attraversamento di un albero

- Stampare gli elementi in successione ordinata:

1. Stampa il sottoalbero sinistro
2. Stampa i dati
3. Stampa il sottoalbero destro

Esempio

- Facciamo una prova con l'albero della Figura 10.
- L'algoritmo ci dice di

1. Stampare il sottoalbero sinistro di `Juliet`; vale a dire `Dick` e i suoi discendenti
2. Stampare `Juliet`
3. Stampare il sottoalbero destro di `Juliet`; vale a dire `Tom` e i suoi discendenti

Esempio

- Come fate a stampare il sottoalbero che ha come radice `Dick`?

1. Stampate il sottoalbero sinistro di `Dick`: non c'è niente da stampare.
2. Stampate `Dick`.
3. Stampate il sottoalbero destro di `Dick`, vale a dire, `Harry`.

Esempio

- Visualizza

```
Dick
Harry
Juliet
Romeo
Tom
```

- L'albero viene stampato in successione ordinata.

Il metodo `print` della classe `BinarySearchTree`

```
public class BinarySearchTree
{
   . . .
   public void print()
   {
      if (root != null)
         root.printNodes();

      System.out.println();
   }
   . . .
}
```

Il metodo `printNode` della classe `Node`

```
private class Node
{
   . . .
   public void printNodes()
   {
      if (left != null)
         left.printNodes();
      System.out.println(data);
      if (right != null)
         right.printNodes();
   }
   . . .
}
```

Attraversamento di un albero

- Strategie di attraversamento di alberi
- *attraversamento in ordine anticipato* (“preorder traversal”)
- *attraversamento in ordine posticipato* (“postorder traversal”)
- *attraversamento in ordine simmetrico* (“inorder traversal”)

Attraversamento in ordine anticipato

- Si visita la radice
- Si visita il suo sottoalbero sinistro
- Si visita il suo sottoalbero destro

Attraversamento in ordine simmetrico

- Si visita il suo sottoalbero sinistro
- Si visita la radice
- Si visita il suo sottoalbero destro

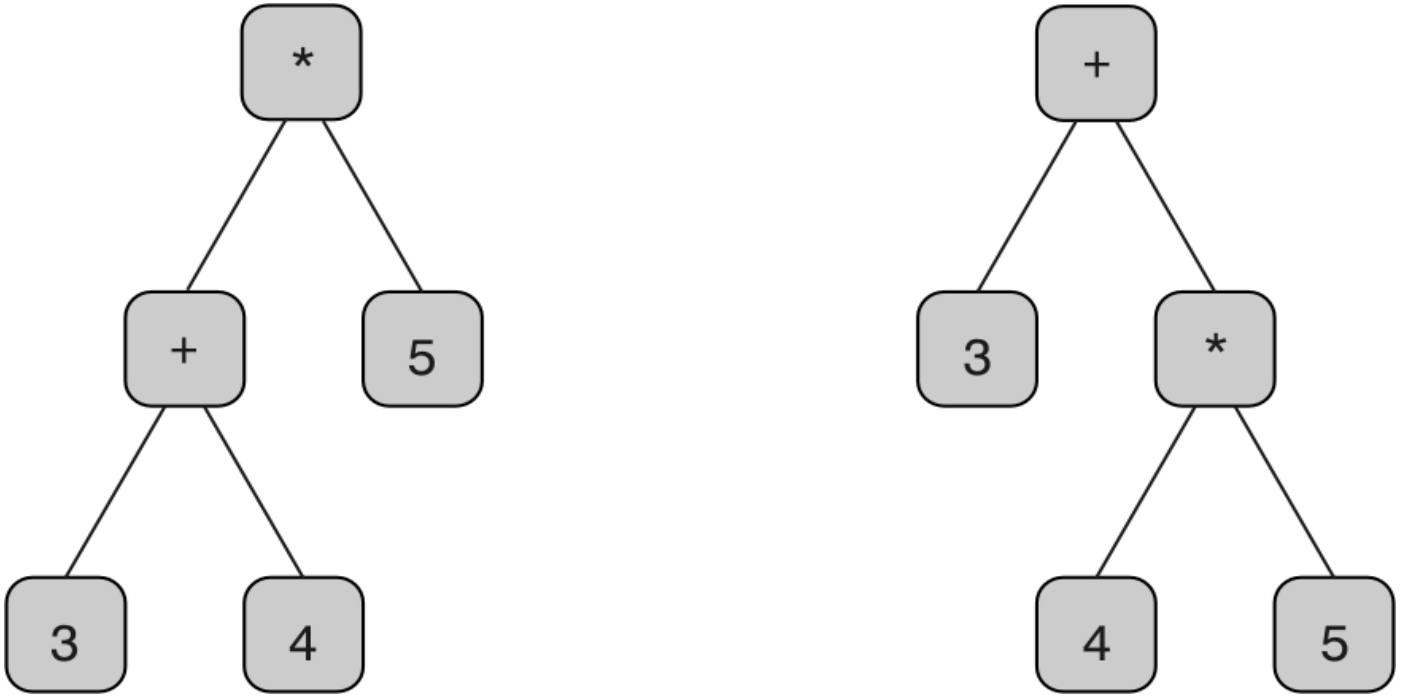
Attraversamento in ordine posticipato

- Si visita il sottoalbero sinistro della radice
- Si visita il suo sottoalbero destro della radice
- Si visita la radice

Alberi che rappresentano un'espressione

- L'attraversamento in ordine posticipato di un albero di espressione fornisce le istruzioni necessarie per valutare l'espressione stessa mediante una calcolatrice con funzionamento a stack

Figura 14



Attraversamento di un albero

- Per il primo albero (3 + 4) * 5 si ha:

3 4 + 5 *

- Per il secondo 3 + 4 * 5 si ottiene:

3 4 5 * +

Una calcolatrice con funzionamento *a stack*

- Un numero significa:

• Inserisci il numero sulla pila.

- Un operatore significa:

• Estrai dalla pila i due numeri che si trovano in cima.
• Applica l'operatore a tali due numeri.
• Inserisci nella pila il risultato.

Una calcolatrice con funzionamento *a stack*

- Per la valutazione di espressioni aritmetiche.
- Si trasforma l'espressione in un albero
- Si effettua un attraversamento in ordine posticipato dell'albero di espressione
- Si eseguono le operazioni nell'ordine indicato

- Il risultato è il valore dell'espressione

Una calcolatrice con funzionamento *a stack*

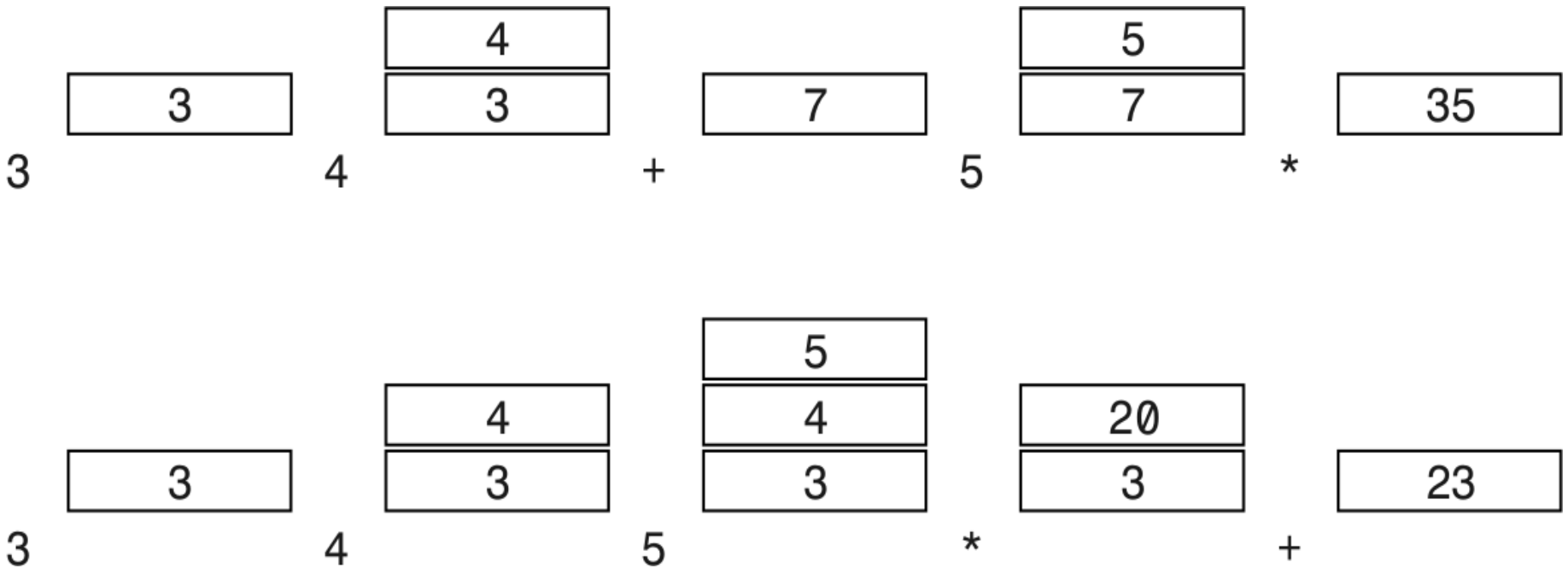


Figura 15

Usare insiemi e mappe realizzati con alberi

- Entrambe le classi `HashSet` e `TreeSet` realizzano l'interfaccia `Set`
- Se i vostri oggetti hanno una buona funzione di hash, l'uso di una tabella hash è solitamente più veloce degli algoritmi basati su alberi
- Ma gli alberi bilanciati usati nella classe `TreeSet` possono *garantire* prestazioni ragionevoli
- La classe `HashSet` si affida completamente alla funzione di hash

Usare un oggetto di tipo `TreeSet`

- I vostri oggetti devono appartenere a una classe che realizza l'interfaccia `Comparable`
- Oppure dovete fornire un oggetto di tipo `Comparator`

Usare un oggetto di tipo `TreeMap`

- Le chiavi devono appartenere a una classe che realizza l'interfaccia `Comparable`
- Oppure dovete fornire un oggetto di tipo `Comparator`
- Non c'è nessun requisito per i valori

File TreeSetTester.java

```
01: import java.util.Comparator;
02: import java.util.Set;
03: import java.util.TreeSet;
04:
05:
06: /**
07:    Un programma per collaudare un insieme di monete
realizzato con un albero.
08: */
09: public class TreeSetTester
10: {
11:    public static void main(String[] args)
12:    {
13:       Coin coin1 = new Coin(0.25, "quarter");
14:       Coin coin2 = new Coin(0.25, "quarter");
15:       Coin coin3 = new Coin(0.01, "penny");
16:       Coin coin4 = new Coin(0.05, "nickel");
17:
```

Continua…

File TreeSetTester.java

```
18:         class CoinComparator implements Comparator<Coin>
19:         {
20:            public int compare(Coin first, Coin second)
21:            {
22:               if (first.getValue()
                     < second.getValue()) return -1;
23:               if (first.getValue()
                        == second.getValue()) return 0;
24:               return 1;
25:            }
26:         }
27:
28:         Comparator<Coin> comp = new CoinComparator();
29:         Set<Coin> coins = new TreeSet<Coin>(comp);
30:         coins.add(coin1);
31:         coins.add(coin2);
32:         coins.add(coin3);
33:         coins.add(coin4);
```

Continua…

File TreeSetTester.java

```
34:          for (Coin c : coins)
35:             System.out.print(c.getValue() + " ");
36:          System.out.println("Expected: 0.01 0.05 0.25");
37:       }
38: }
```

File `TreeSetTester.java`

- Visualizza

```
Coin[value=0.01,name=penny]
Coin[value=0.05,name=nickel]
Coin[value=0.25,name=quarter]
```

Code prioritarie

- La *coda prioritaria*, funge da contenitore di elementi, a ciascuno dei quali viene assegnata una *priorità*
- Un tipico esempio di coda prioritaria è un insieme di richieste di lavori da svolgere, alcuni dei quali possono essere più urgenti di altri
- Quando si elimina un elemento da una coda prioritaria viene rimosso l'elemento con la priorità più elevata
- Per convenzione, solitamente si assegnano valori più bassi alle priorità più alte, con la priorità di valore 1 che indica la priorità massima
- La libreria standard di Java mette a disposizione una classe `PriorityQueue`
- Un'altra struttura dati, chiamata "heap", è ancora più adatta per realizzare code prioritarie

Esempio

- Considerate il codice seguente

```
PriorityQueue<WorkOrder> q = new PriorityQueue<WorkOrder>;
q.add(new WorkOrder(3, "Shampoo carpets"));
q.add(new WorkOrder(1, "Fix overflowing sink"));
q.add(new WorkOrder(2, "Order cleaning supplies"));
```

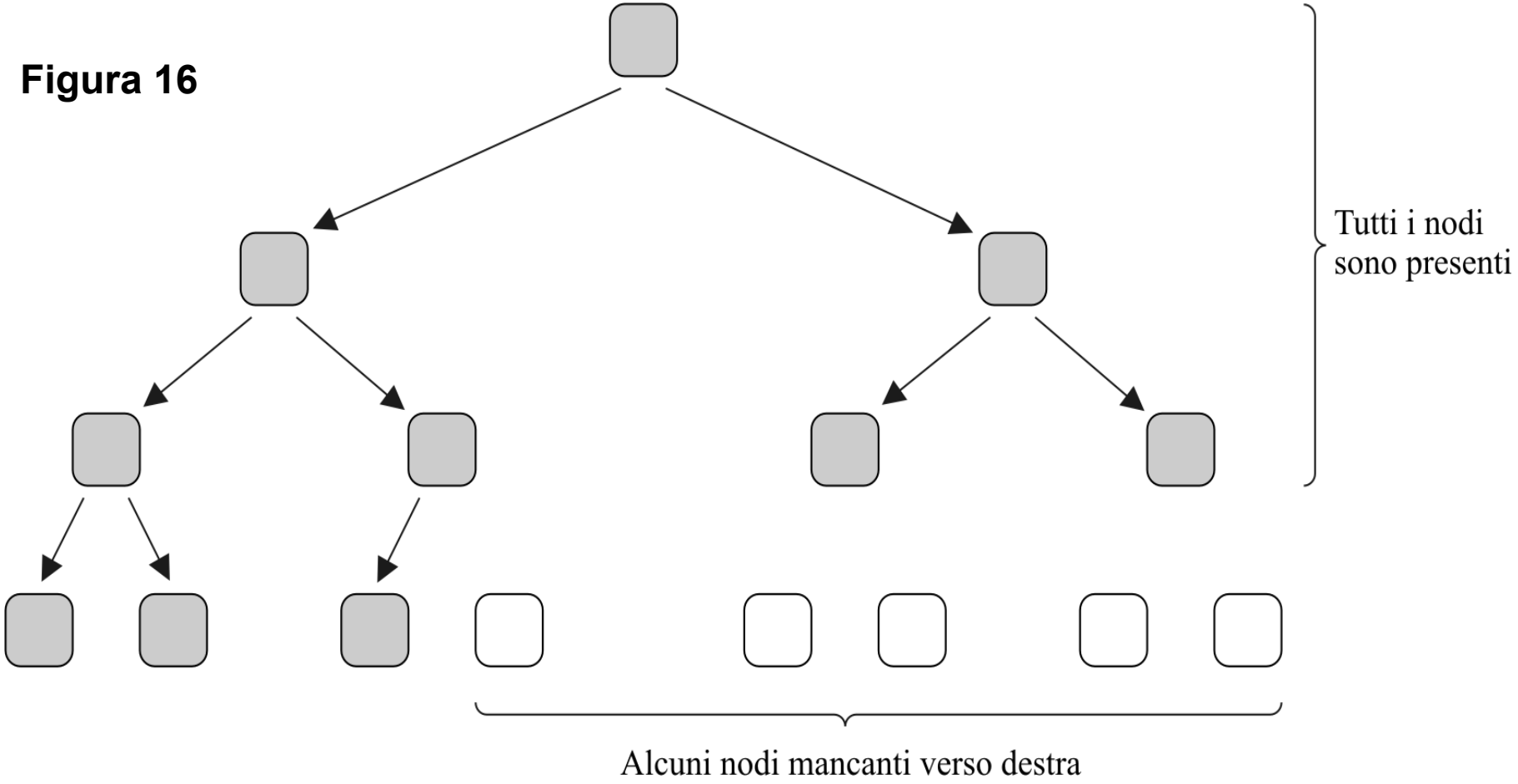
- Quando si invoca `q.remove()` per la prima volta, viene estratto dalla coda l'ordine di lavoro con priorità 1.
- La successiva invocazione di `q.remove()` estrae l'ordine di lavoro di priorità più elevata tra quelli rimasti nella coda, che nel nostro esempio è l'ordine di lavoro con priorità 2

Heap

- Uno *heap* (“mucchio”, chiamato anche, per maggior chiarezza, *min-heap*) è un albero binario dotato di due speciali proprietà.
- Uno heap è *quasi completo*: a ogni livello sono presenti tutti i nodi, tranne al livello più basso dell’albero dove può mancare qualche nodo nella parte destra
- L’albero soddisfa la “proprietà di heap”: ogni nodo memorizza un valore che non è superiore ai valori memorizzati nei suoi discendenti
- È facile verificare che la proprietà di heap garantisce che nella radice sia memorizzato il valore minimo.

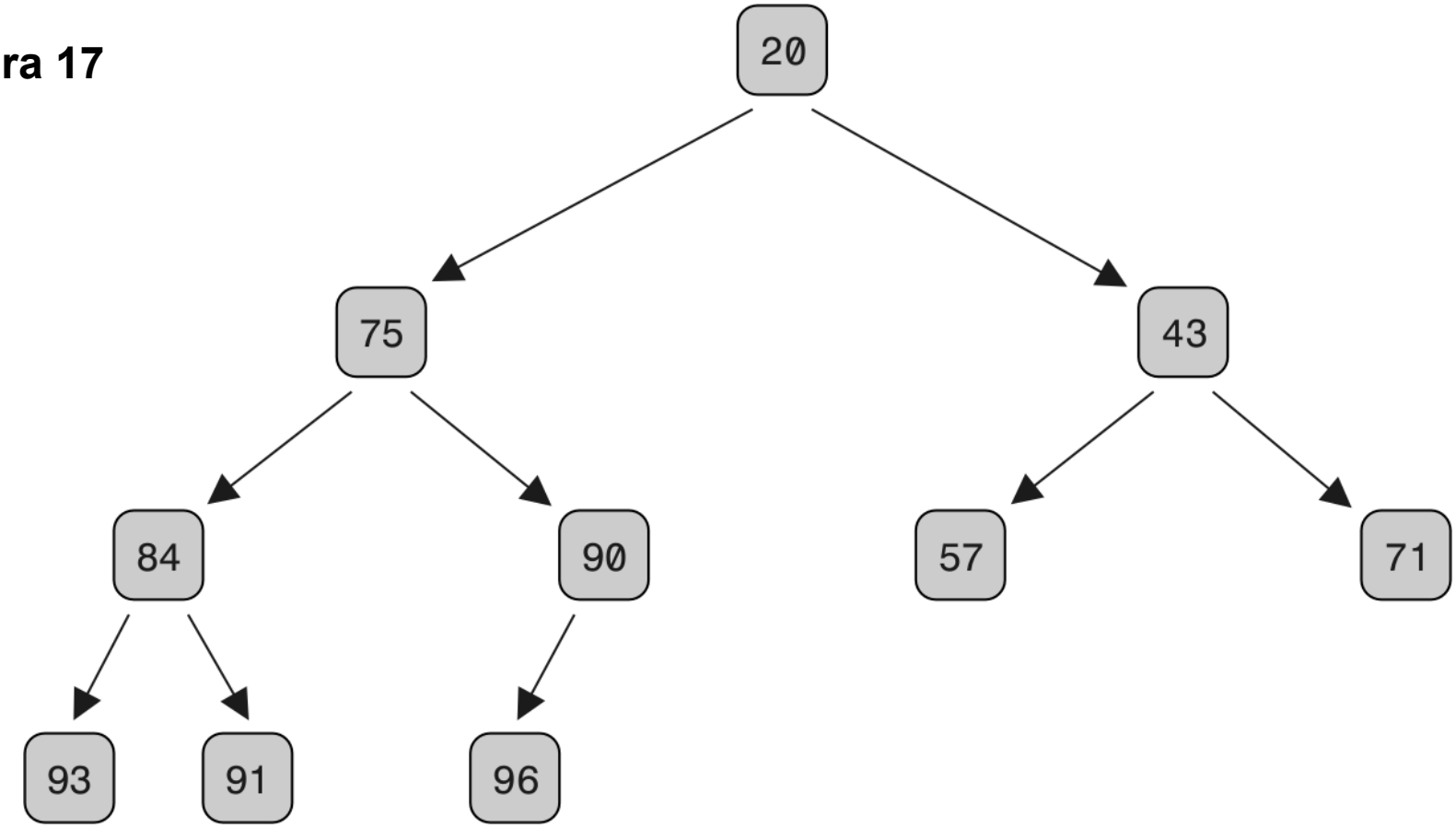
Un albero quasi completo

Figura 16



Uno Heap

Figura 17



Differenze tra uno heap e un albero di ricerca binario

- La forma di uno heap è molto regolare
- Gli alberi di ricerca binari possono avere forme arbitrarie
- In uno heap, sia il sottoalbero di sinistra che il sottoalbero di destra contengono elementi di valore maggiore o uguale all'elemento radice
- In un albero di ricerca binario gli elementi minori sono memorizzati nel sottoalbero sinistro e gli elementi maggiori sono memorizzati nel sottoalbero destro

Inserimento di un elemento in uno heap

1. Aggiungete alla fine dello heap un nodo vuoto.

2. Se il genitore del nodo vuoto ha un valore maggiore dell'elemento da inserire, "declassatelo", cioè spostate nel nodo vuoto il valore contenuto nel genitore
   - Rendete così vuoto il nodo genitore
   - Ripetete il declassamento fintanto ché il genitore del nodo vuoto ha un valore maggiore dell'elemento da inserire
3. A questo punto: o il nodo vuoto si trova nella radice, oppure il genitore del nodo vuoto ha un valore inferiore dell'elemento da inserire. Inserite l'elemento nel nodo vuoto

Inserimento di un elemento in uno heap

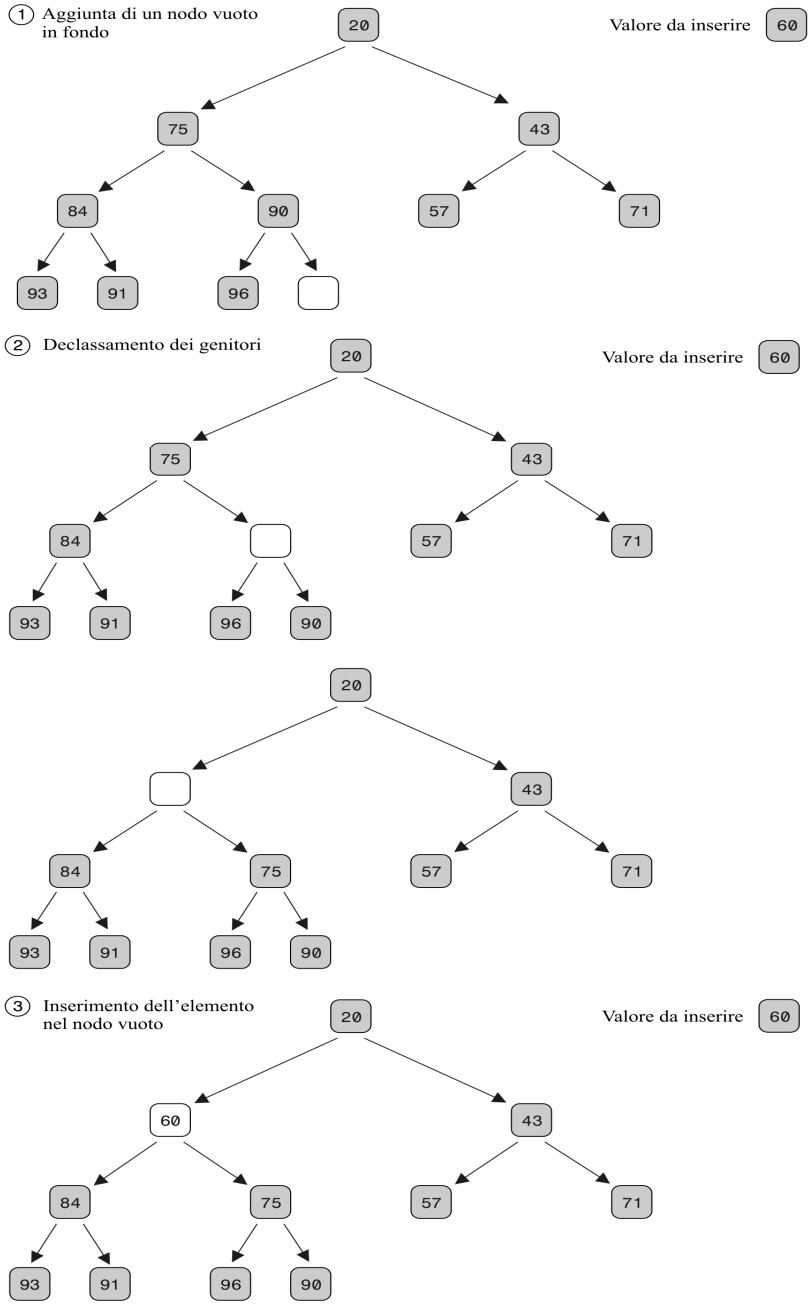


Figura 18

Eliminazione del valore minimo da uno heap

1. Eliminate il valore presente nel nodo radice
2. Trasferite nella radice il valore presente nell'ultimo nodo dello heap ed eliminate tale nodo. A questo punto può darsi che il nodo radice non rispetti la proprietà di heap, perché uno dei suoi figli (o entrambi) può avere un valore inferiore.
3. "Promuovete" il figlio della radice che ha il valore minore, tra i due, come si può vedere nella Figura 19: ora il nodo radice rispetta nuovamente la proprietà di heap. Ripetete questo procedimento con il figlio che è stato appena declassato, cioè promuovete il suo figlio di valore minore; continuate fino a quando il figlio declassato non ha figli di valore inferiore. A questo punto la proprietà di heap è nuovamente soddisfatta. Questo procedimento viene chiamato "sistemazione dello heap" (*fixing the heap*).

Eliminazione del valore minimo da uno heap

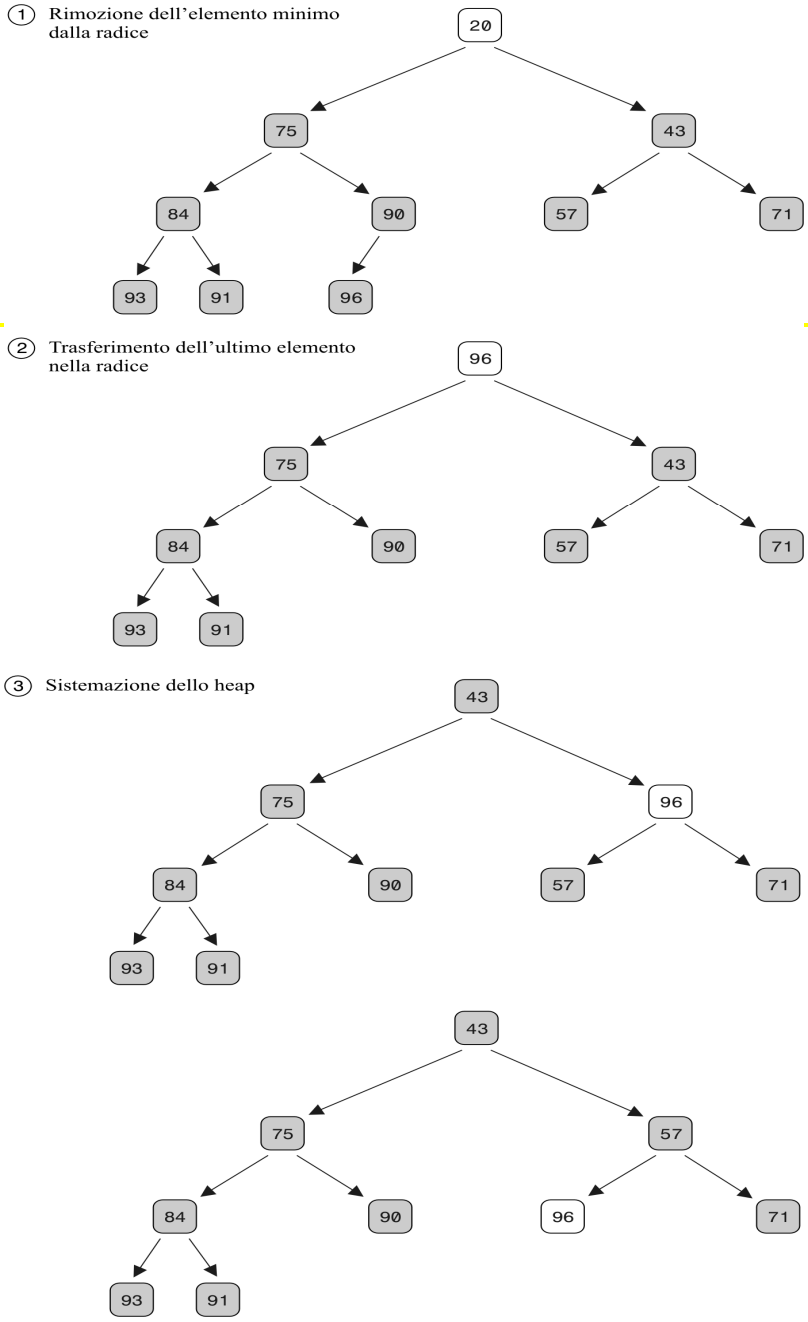


Figura 19

Efficienza dello Heap

- Le operazioni di inserimento e rimozione visitano al massimo *h* nodi
- *h* è l'altezza dell'albero
- se *n* è il numero di elementi, allora:

$$2^{h-1} \leq n < 2^h$$

oppure

$$h - 1 \leq \log_2(n) < h$$

- In uno heap, l'inserimento e la rimozione di un elemento sono operazioni O(log n)
- La disposizione regolare dei nodi di uno heap rende possibile una loro efficiente memorizzazione in un array

Memorizzazione di uno heap in un array

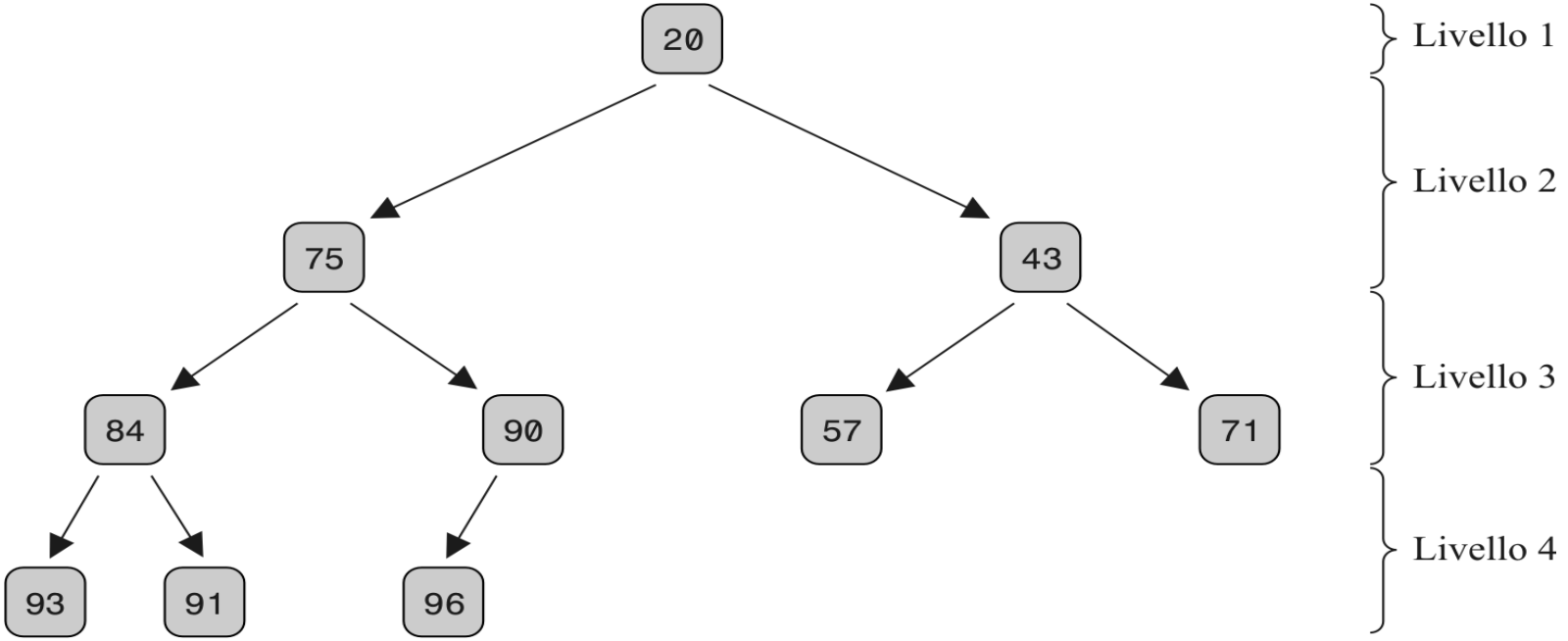


Figura 20

File `MinHeap.java`

```
001: import java.util.*;
002:
003: /**
004:    Classe che realizza un min-heap.
005: */
006: public class MinHeap
007: {
008:    /**
009:       Costruisce uno heap vuoto.
010:    */
011:    public MinHeap()
012:    {
013:       elements = new ArrayList<Comparable>();
014:       elements.add(null);
015:    }
016:
```

Continua…

File `MinHeap.java`

```
017:    /**
018:       Aggiunge un nuovo elemento allo heap.
019:       @param newElement l'elemento da aggiungere
020:    */
021:    public void add(Comparable newElement)
022:    {
023:       // aggiunge una nuova foglia
024:       elements.add(null);
025:       int index = elements.size() - 1;
026:
027:// declassa i genitori che hanno valore maggiore del nuovo
   elemento
028:       while (index > 1
029:             && getParent(index).compareTo(newElement) >
   0)
030:       {
031:          elements.set(index, getParent(index));
032:          index = getParentIndex(index);
033:       }
```

Continua…

File MinHeap.java

```
034:
035:        // memorizza il nuovo elemento nel nodo vuoto
036:        elements.set(index, newElement);
037:     }
038:
039:     /**
040:        Restituisce l'elemento minimo presente nello heap.
041:                   @return l'elemento minimo
042:     */
043:     public Comparable peek()
044:     {
045:        return elements.get(1);
046:     }
047:
048:     /**
049:        Elimina dallo heap l'elemento minimo.
050:        @return l'elemento minimo
051:     */
```

Continua…

File MinHeap.java

```
052:    public Comparable remove()
053:    {
054:       Comparable minimum = elements.get(1);
055:
056:       // elimina l'ultimo elemento
057:       int lastIndex = elements.size() - 1;
058:       Comparable last = elements.remove(lastIndex);
059:
060:       if (lastIndex > 1)
061:       {
062:          elements.set(1, last);
063:          fixHeap();
064:       }
065:
066:       return minimum;
067:    }
068:
```

Continua…

File MinHeap.java

```
069:    /**
070:       Ripristina la proprietà di heap, a condizione che
071:       soltanto il nodo radice la violi.
072:    */
073:    private void fixHeap()
074:    {
075:       Comparable root = elements.get(1);
076:
077:       int lastIndex = elements.size() - 1;
078:       // promuove i figli della radice rimossa finché
              sono maggiori dell'ultimo elemento
079:
080:       int index = 1;
081:       boolean more = true;
082:       while (more)
083:       {
084:          int childIndex = getLeftChildIndex(index);
085:          if (childIndex <= lastIndex)
086:          {
```

Continua…

File MinHeap.java

```
087:             // identifica il figlio minore
088:
089:             // considera prima il figlio sinistro
090:             Comparable child = getLeftChild(index);
091:
092:             // usa invece il figlio destro se è minore
093:             if (getRightChildIndex(index) <= lastIndex
094:                   && getRightChild(index).compareTo(child) <)
095:             {
096:                childIndex = getRightChildIndex(index);
097:                child = getRightChild(index);
098:             }
099:
100:             // verifica se il figlio maggiore è minore della
                 radice
101:             if (child.compareTo(root) < 0)
102:             {
103:                // promuove il figlio
```

Continua…

File `MinHeap.java`

```
104:                elements.set(index, child);
105:                index = childIndex;
106:             }
107:             else
108:             {
109:                // la radice è minore di entrambi i figli
110:                more = false;
111:             }
112:          }
113:          else
114:          {
115:             // non ci sono figli
116:             more = false;
117:          }
118:       }
119:
120:       // memorizza nel nodo vuoto l'elemento presente
   nella radice
121:       elements.set(index, root);
122:    }
```

Continua…

File MinHeap.java

```
123:
124:    /**
125:       Restituisce il numero di elementi presenti nello
   heap.
126:    */
127:    public int size()
128:    {
129:       return elements.size() - 1;
130:    }
131:
132:    /**
133:       Restituisce l'indice del figlio sinistro.
134:       @param index l'indice di un nodo dello heap
135:       @return l'indice del figlio sinistro del nodo dato
136:    */
137:    private static int getLeftChildIndex(int index)
138:    {
139:       return 2 * index;
140:    }
```

Continua…

File MinHeap.java

```
141:
142:    /**
143:       Restituisce l'indice del figlio destro.
144:       @param index l'indice di un nodo dello heap
145:       @return l'indice del figlio destro del nodo dato
146:    */
147:    private static int getRightChildIndex(int index)
148:    {
149:       return 2 * index + 1;
150:    }
151:
152:    /**
153:       Restituisce l'indice del genitore.
154:       @param index l'indice di un nodo dello heap
155:       @return l'indice del genitore del nodo dato
156:    */
```

Continua…

File MinHeap.java

```
157:    private static int getParentIndex(int index)
158:    {
159:       return index / 2;
160:    }
161:
162:    /**
163:       Restituisce il valore del figlio sinistro.
164:       @param index l'indice di un nodo dello heap
165:       @return il valore del figlio sinistro del nodo dato
166:    */
167:    private Comparable getLeftChild(int index)
168:    {
169:       return elements.get(2 * index);
170:    }
171:
172:    /**
173:       Restituisce il valore del figlio destro.
174:       @param index l'indice di un nodo dello heap
```

Continua…

File MinHeap.java

```
175:          @return il valore del figlio destro del nodo dato
176:       */
177:       private Comparable getRightChild(int index)
178:       {
179:          return elements.get(2 * index + 1);
180:       }
181:
182:       /**
183:          Restituisce il valore del genitore.
184:          @param l'indice di un nodo dello heap
185:          @return il valore del genitore del nodo dato
186:       */
187:       private Comparable getParent(int index)
188:       {
189:          return elements.get(index / 2);
190:       }
191:
192:       private ArrayList<Comparable> elements;
193: }
```

File HeapDemo.java

```
01: /**
02:    Questo programma mostra come realizzare una
    coda prioritaria usando uno heap.
03: */
04: public class HeapDemo
05: {
06:    public static void main(String[] args)
07:    {
08:       MinHeap q = new MinHeap();
09:       q.add(new WorkOrder(3, "Shampoo carpets"));
10:       q.add(new WorkOrder(7, "Empty trash"));
11:       q.add(new WorkOrder(8, "Water plants"));
12:       q.add(new WorkOrder(10, "Remove pencil sharpener shavings"));
13:       q.add(new WorkOrder(6, "Replace light bulb"));
14:       q.add(new WorkOrder(1, "Fix broken sink"));
15:       q.add(new WorkOrder(9, "Clean coffee maker"));
16:       q.add(new WorkOrder(2, "Order cleaning supplies"));
17:
```

Continua…

File HeapDemo.java

```
18:         while (q.size() > 0)
19:             System.out.println(q.remove());
20:     }
21: }
```

File WorkOrder.java

```
01: /**
02:    Questa classe incapsula un ordine di lavoro avente una
   priorità.
03: */
04: public class WorkOrder implements Comparable
05: {
06:    /**
07:       Costruisce un ordine di lavoro con una data priorità
       e descrizione.
08:       @param aPriority la priorità dell'ordine di lavoro
09:       @param aDescription la descrizione dell'ordine di
   lavoro
10:    */
11:    public WorkOrder(int aPriority, String aDescription)
12:    {
13:       priority = aPriority;
14:       description = aDescription;
15:    }
16:
```

Continua…

File WorkOrder.java

```
17:     public String toString()
18:     {
19:        return "priority=" + priority + ", description="
              + description;
20:     }
21:
22:     public int compareTo(Object otherObject)
23:     {
24:        WorkOrder other = (WorkOrder) otherObject;
25:        if (priority < other.priority) return -1;
26:        if (priority > other.priority) return 1;
27:        return 0;
28:     }
29:
30:     private int priority;
31:     private String description;
32: }
```

File `WorkOrder.java`

Visualizza:

```
priority=1, description=Fix broken sink
priority=2, description=Order cleaning supplies
priority=3, description=Shampoo carpets
priority=6, description=Replace light bulb
priority=7, description=Empty trash
priority=8, description=Water plants
priority=9, description=Clean coffee maker
priority=10, description=Remove pencil sharpener shavings
```

L'algoritmo `Heapsort`

- L'algoritmo `heapsort` è basato sull'inserimento di elementi in uno heap e sulla loro successiva rimozione ordinata
- Questo algoritmo ha prestazioni $O(n \log n)$: ciascun inserimento e rimozione è un'operazione $O(\log n)$ e tali operazioni vengono ripetute n volte ciascuna, una volta per ogni elemento della sequenza che deve essere ordinata

L'algoritmo `Heapsort`

- L'algoritmo può essere reso un po' più efficiente
- Iniziamo con una sequenza di valori inseriti in un array
- Trasformiamo in heap dapprima dei piccoli sottoalberi, per poi passare ad alberi più grandi.
- Gli alberi di dimensione 1 sono automaticamente degli heap
- Possiamo iniziare la procedura con i sottoalberi le cui radici si trovano nel penultimo livello dell'albero
- Il metodo `fixHeap` generalizzato trasforma in heap un sottoalbero avente radice in un elemento di indice dato:

```
void fixHeap(int rootIndex, int lastIndex)
```

Trasformazione di un albero in uno heap

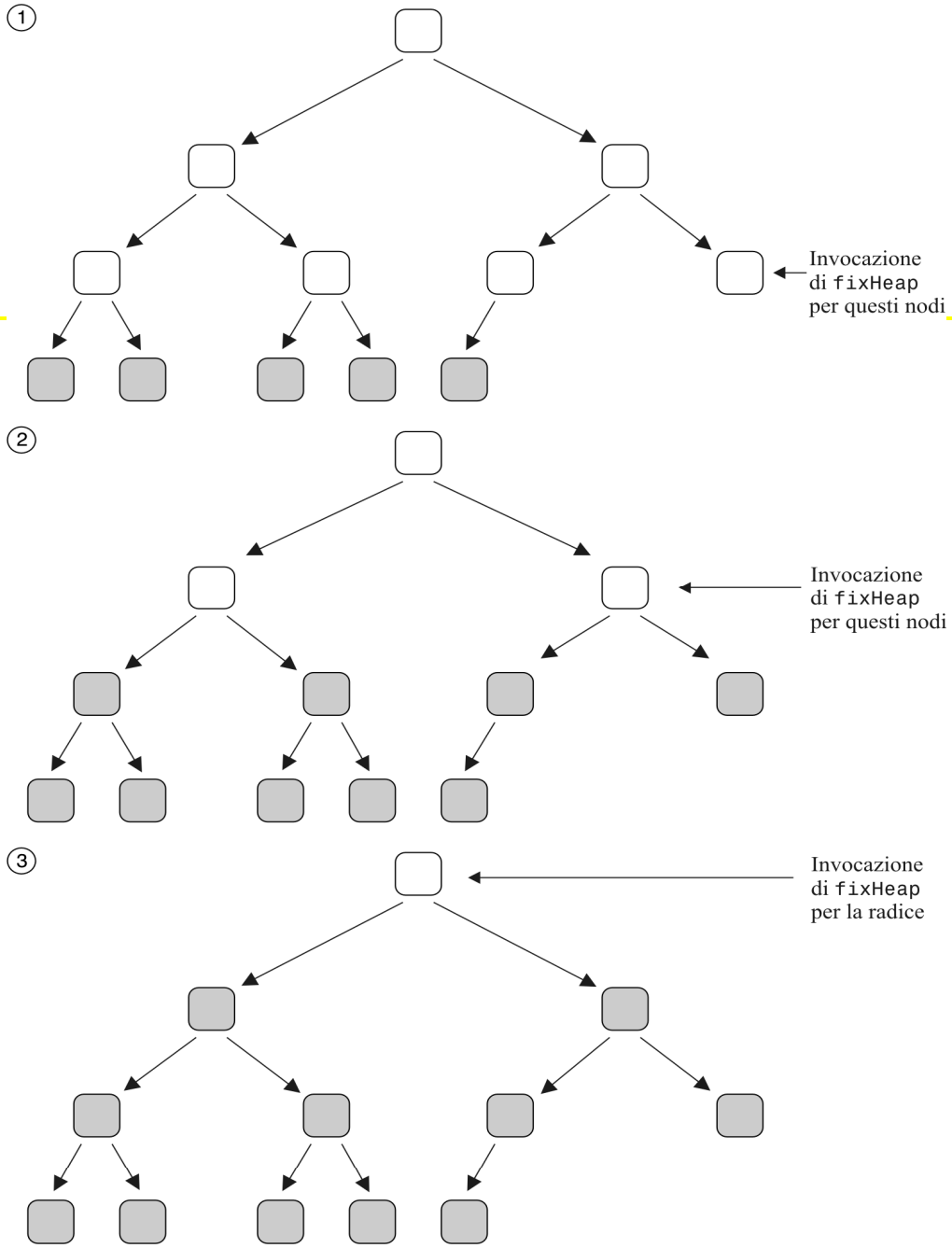


Figura 21

L'algoritmo `Heapsort`

- Dopo aver trasformato l'array in heap, ne eliminiamo ripetutamente l'elemento radice
- *Scambieremo* l'elemento radice con l'ultimo elemento dell'albero, riducendo così la lunghezza "logica" dell'albero
- Di conseguenza, la radice eliminata viene a trovarsi nell'ultima posizione dell'array, che non è più utilizzata dallo heap
- Siamo in grado di usare il medesimo array sia per contenere lo heap (che, a ogni passo, diventa più piccolo) sia la sequenza ordinata (che, al contrario, diventa sempre più lunga)
- Usando un min-heap, la sequenza ordinata viene memorizzata al contrario, con l'elemento più piccolo che viene a trovarsi alla fine dell'array

Utilizzo di `Heapsort` per l'ordinamento di un array

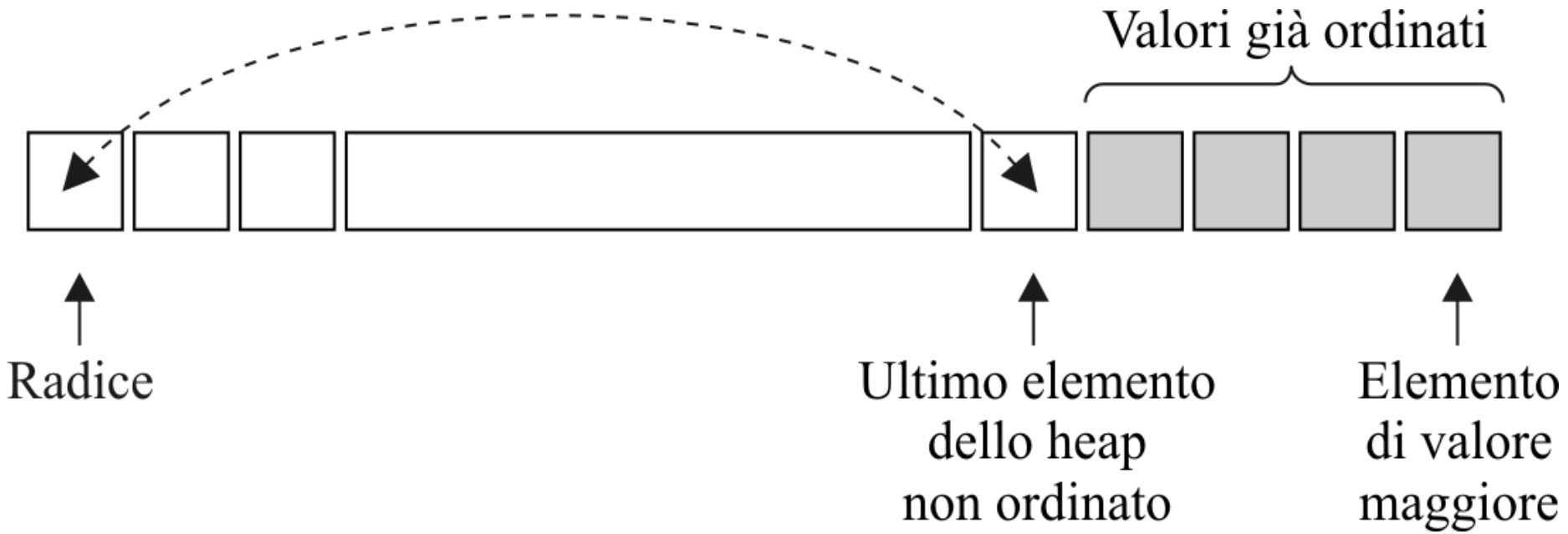


Figura 22

File HeapSorter.java

```
001: /**
002:    Questa classe applica l'algoritmo heapsort
       per ordinare un array.
003: */
004: public class HeapSorter
005: {
006:    /**
007:       Costruisce un oggetto che ordina un array usando
           l'algoritmo heapsort.
008:       @param anArray un array di numeri interi
009:    */
010:    public HeapSorter(int[] anArray)
011:    {
012:       a = anArray;
013:    }
014:
015:    /**
016:       Ordina l'array gestito da questo oggetto.
017:    */
```

File HeapSorter.java

```
018:    public void sort()
019:    {
020:       int n = a.length - 1;
021:       for (int i = (n - 1) / 2; i >= 0; i--)
022:          fixHeap(i, n);
023:       while (n > 0)
024:       {
025:          swap(0, n);
026:          n--;
027:          fixHeap(0, n);
028:       }
029:    }
030:
031:    /**
032:       Rende vera la proprietà di heap per un sottoalbero,
033: nell'ipotesi che i figli della sua radice la rispettino già.
```

File HeapSorter.java

```
034:        @param rootIndex l'indice della radice del sottoalbero
            da sistemare
035:        @param lastIndex l'ultimo indice valido all'interno
            dell'albero che contiene il sottoalbero da sistemare
037:     */
038:     private void fixHeap(int rootIndex, int lastIndex)
039:     {
040:        // elimina la radice
041:        int rootValue = a[rootIndex];
042:
043:        // promuove i figli di valore maggiore della radice
044:
045:        int index = rootIndex;
046:        boolean more = true;
047:        while (more)
048:        {
049:           int childIndex = getLeftChildIndex(index);
050:           if (childIndex <= lastIndex)
```

File HeapSorter.java

```
051:          {
052:             // usa invece il figlio destro se è maggiore
053:             int rightChildIndex = getRightChildIndex(index);
054:             if (rightChildIndex <= lastIndex
055:                   && a[rightChildIndex] > a[childIndex])
056:             {
057:                childIndex = rightChildIndex;
058:             }
059:
060:             if (a[childIndex] > rootValue)
061:             {
062:                // promuove il figlio
063:                a[index] = a[childIndex];
064:                index = childIndex;
065:             }
066:             else
067:             {
```

File HeapSorter.java

```
068:    // il valore della radice è maggiore di quello di
   entrambi i figli
069:                   more = false;
070:                }
071:             }
072:             else
073:             {
074:                // non ci sono figli
075:                more = false;
076:             }
077:          }
078:
079:          // memorizza nel nodo vuoto il valore della radice
080:          a[index] = rootValue;
081:       }
082:
```

File HeapSorter.java

```
083:    /**
084:       Scambia due valori nell'array.
085:       @param i la prima posizione da scambiare
086:       @param j la seconda posizione da scambiare
087:    */
088:    private void swap(int i, int j)
089:    {
090:       int temp = a[i];
091:       a[i] = a[j];
092:       a[j] = temp;
093:    }
094:
095:    /**
096:       Restituisce l'indice del figlio sinistro.
097:       @param index l'indice di un nodo dello heap
098:       @return l'indice del nodo sinistro del nodo
   indicato
099:    */
```

File HeapSorter.java

```
100:    private static int getLeftChildIndex(int index)
101:    {
102:       return 2 * index + 1;
103:    }
104:
105:    /**
106:       Restituisce l'indice del figlio destro.
107:       @param index l'indice di un nodo dello heap
108:       @return l'indice del nodo destro del nodo indicato
109:    */
110:    private static int getRightChildIndex(int index)
111:    {
112:       return 2 * index + 2;
113:    }
114:
115:    private int[] a;
116: }
```