


# Programmazione II

A.A. 2022-23

Prof. Maria Tortorella

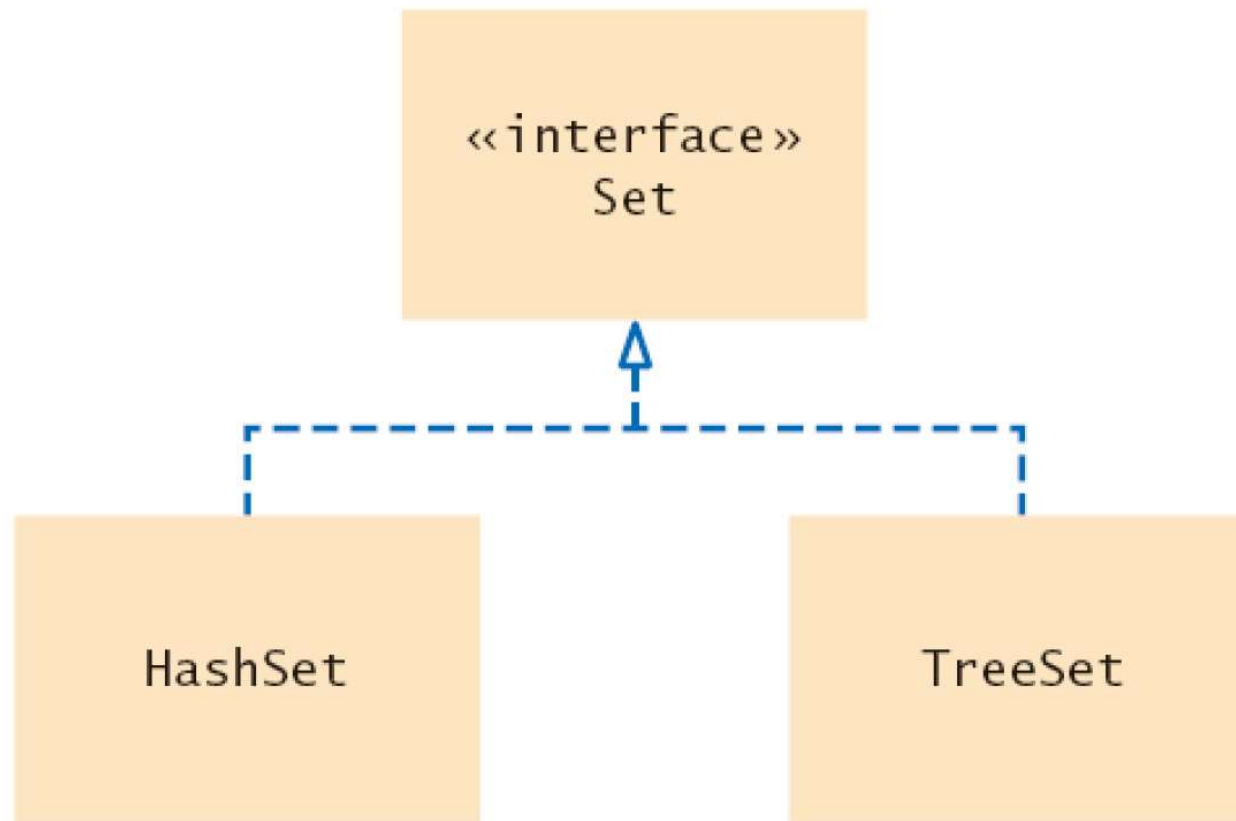
- 
- Set e Map
  - Hash tables e funzioni di hash
    - La gestione delle collisioni
  - Alberi
    - Alberi binari di ricerca
    - Visite

# Sets

---

- **Set**: collezione non ordinata di oggetti che sono unici . . .
- La standard Java library fornisce due implementazioni del set, basate su due strutture diverse
  - HashSet
  - TreeSet
- Entrambe implementano la stessa interfaccia **Set**

# Set

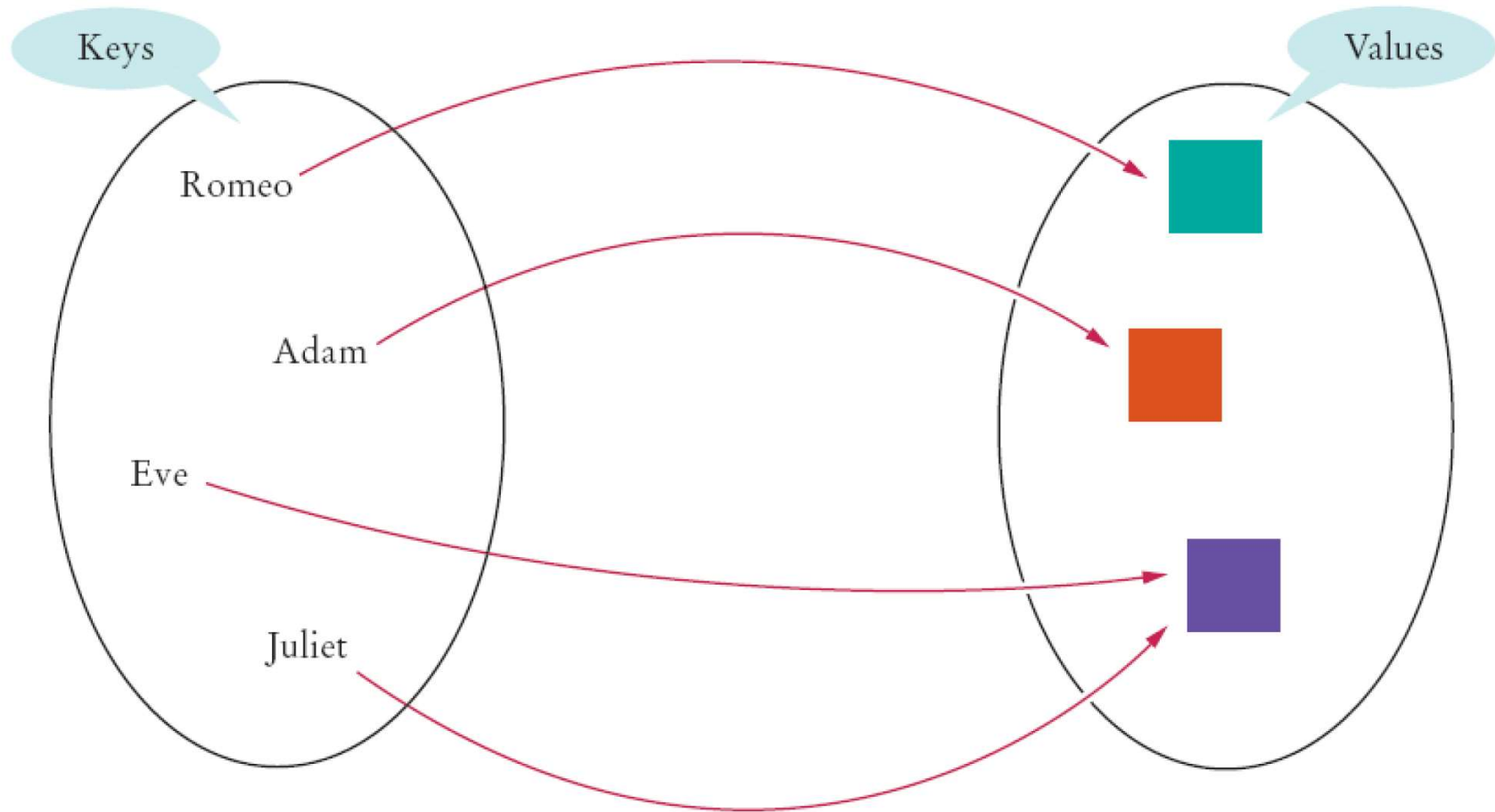


# Maps

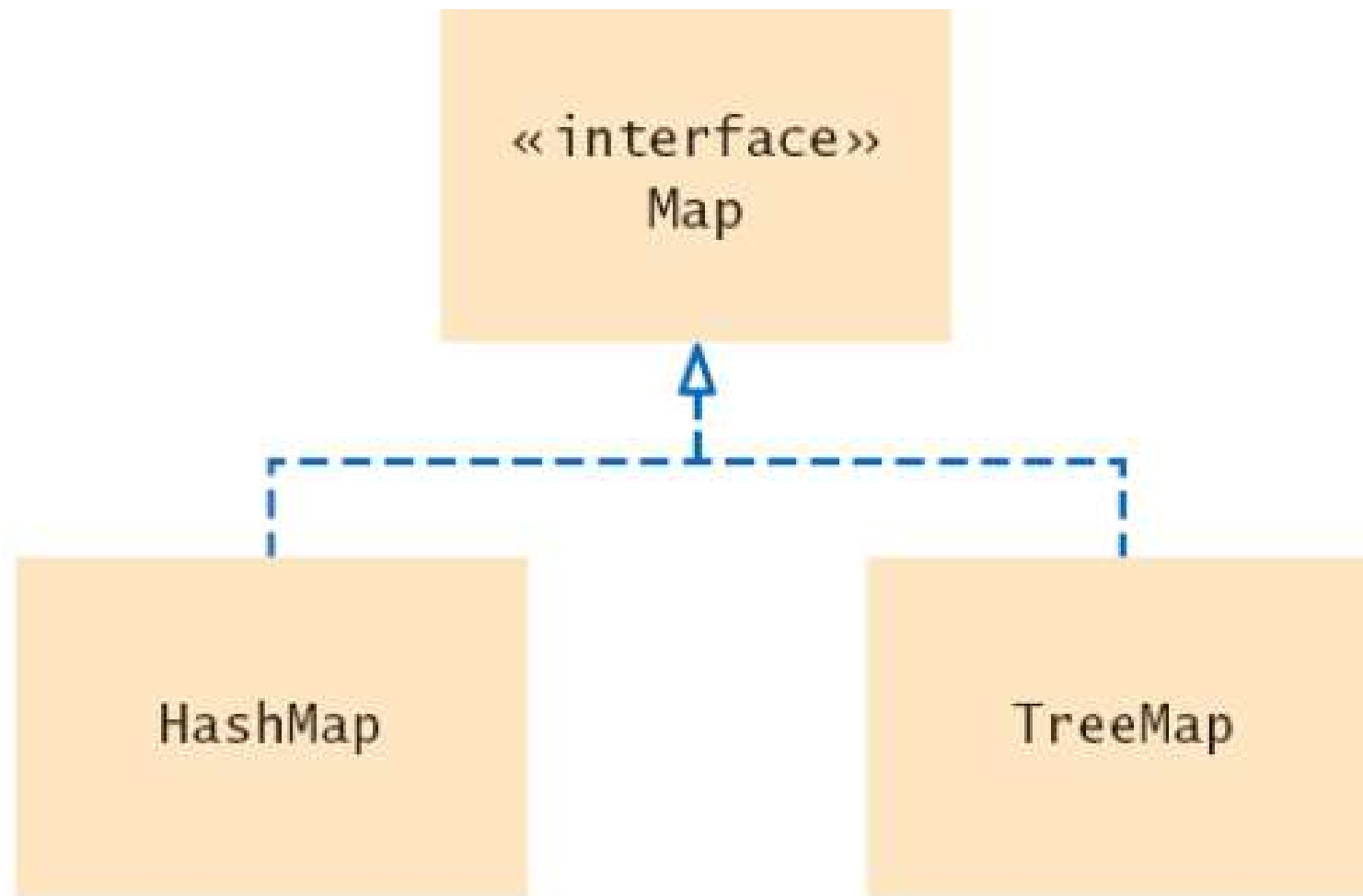
---

- Una mappa memorizza coppie (chiave oggetto)
  - Le chiavi sono uniche
- Come per gli insiemi, i Set:
- Map interface
  - HashMap
  - TreeMap

# Un esempio di una Map



# Map Classes and Interfaces



# Hash Tables

- Hashing: una tecnica di memorizzazione che consente di identificare velocemente un oggetto in una struttura, senza necessità di visite sequenziali
- Una *hash table*
  - utilizza esattamente la tecnica di hashing
  - può essere utilizzata come base per l'implementazione di sets e maps
- Una funzione *hash* *calcola* un valore intero (detto *hash code*) a partire da un oggetto

- Calcolare il codice hash di un oggetto  $x$ :

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

- Una buona funzione deve minimizzare le collisioni
  - Stesso codice per oggetti differenti

# Strings e Hash Codes

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

- I valori degli hash code possono essere o troppo alti o troppo bassi

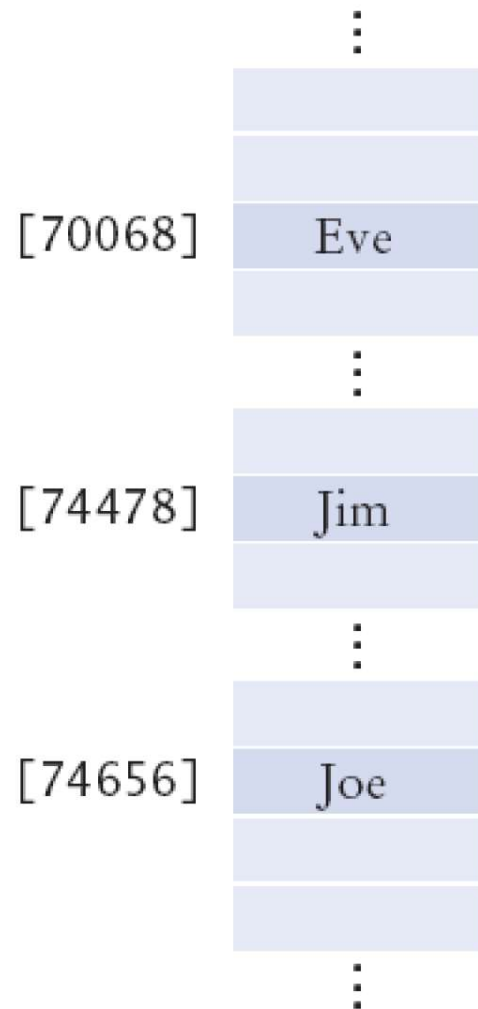


# Hash Table

---

- Una semplice implementazione
  - Creare un array
  - Generare il codice hash degli oggetti
  - Inserire ogni oggetto nella posizione relativa al suo codice
- Verificare la presenza di un oggetto in un set
  - Calcolare il codice hash
  - Verificare se la posizione nell'array è occupata

# Hash Table



- **Problemi**
  - Array virtualmente infinito
  - Non si gestiscono le collisioni
  - Necessità di normalizzazione

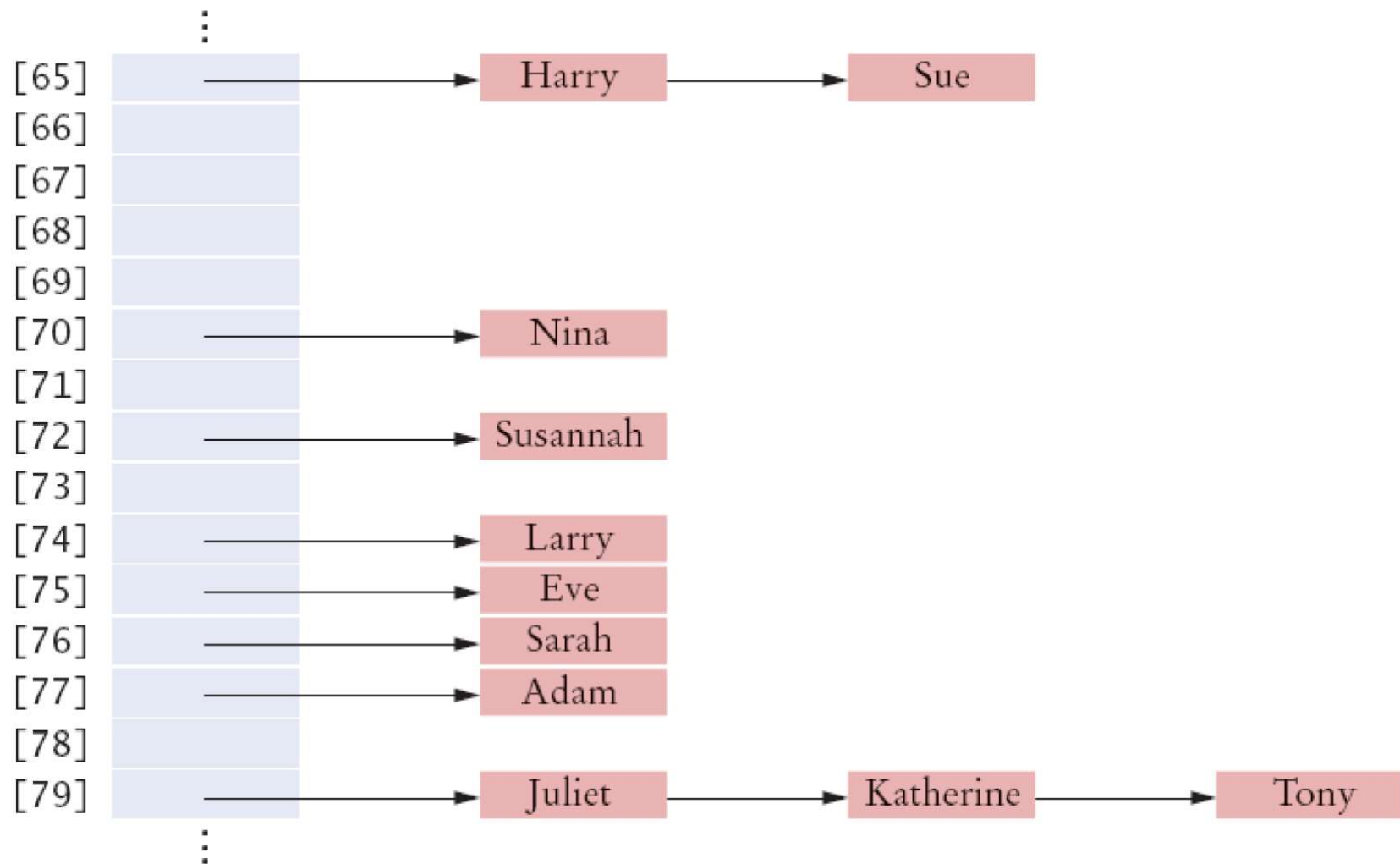
# Soluzione

- Lavorare con una dimensione ragionevole
  - È necessario usare il modulo per identificare l'indice all'interno dell'array

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

- Calcolato in questo modo un valore di h può coincidere per più oggetti: **collisione**
- Ciascuna collisione viene gestita attraverso:
  - Una lista di nodi per gli oggetti multipli
    - *buckets*

# Collisioni e Buckets



# Cercare un oggetto

---

- Si calcola l'indice  $h$  nella hash table
  - hash code
  - modulo
- Si accede alla posizione  $h$
- Si visitano sequenzialmente gli elementi nel bucket di posizione  $h$
- Aggiunta e rimozione: molto simili !

# File HashSet.java

```
001: import java.util.AbstractSet;
002: import java.util.Iterator;
003: import java.util.NoSuchElementException;
004:
005: /**
006:     A hash set stores an unordered collection of
007:     objects, using a hash table.
008: */
009: public class HashSet extends AbstractSet
010: {
011:     /**
012:         Constructs a hash table.
013:         @param bucketsLength length of the buckets array
014:     */
```

# File HashSet.java

```
015:    public HashSet(int bucketsLength)
016:    {
017:        buckets = new Node[bucketsLength];
018:        size = 0;
019:    }
020:
021:    /**
022:        Tests for set membership.
023:        @param x an object
024:        @return true if x is an element of this set
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];
```

# File HashSet.java

```
033:         while (current != null)
034:         {
035:             if (current.data.equals(x)) return true;
036:             current = current.next;
037:         }
038:         return false;
039:     }
040:
041:     /**
042:      Adds an element to this set.
043:      @param x an object
044:      @return true if x is a new object, false if x
045:      was already in the set
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; // Already in the set
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:          Removes an object from this set.
070:          @param x an object
071:          @return true if x was removed from this set,
072:                  false if x was not an element of this set
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] =
087:                                     current.next;
088:         else previous.next = current.next;
089:         size--;
090:         return true;
091:     }
092:     previous = current;
093:     current = current.next;
094: }
095: return false;
096: }
097: /**
098:     Returns an iterator that traverses the elements
099:     of this set.
100:     @param a hash set iterator
101: */
102: public Iterator iterator()
103: {
104:     return new HashSetIterator();
105: }
```

# File HashSet.java

```
105:
106:     /**
107:         Gets the number of elements in this set.
108:         @return the number of elements
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:          * Constructs a hash set iterator that points
128:          * to the first element of the hash set.
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:         // Move to next 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 // Move to next element in 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 SetTester.java

```
01: import java.util.Iterator;
02: import java.util.Set;
03:
04: /**
05:     This program tests the hash set class.
06: */
07: public class SetTester
08: {
09:     public static void main(String[] args)
10:     {
11:         HashSet names = new HashSet(101); //101 is a prime
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");
```

# File SetTester.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 SetTester.java

---

## ➤ Output

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

# Calcolo dei codici hash

- Calcola un intero a partire da un oggetto
  - Minimizzando la possibilità di collisioni

- **String**

- Sommare i codici unicode dei caratteri che la compongono

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

- Insensibile alle permutazioni

# Dalla libreria standard . . .

- Hash function per una stringa  $s$

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

- Esempio "eat"

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

- "tea"

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

# Un metodo `hashCode` per la classe `Coin`

---

- Due variabili:
  - `String` per coin name
  - `double` per coin value
- `String`'s `hashCode` per il nome
- Per il value:
  - Creiamo un oggetto dal valore `Double`
  - Usiamo il metodo `hashCode` method `Double`
- Combiniamo i due codici

# Un 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;
    }
    . . .
}
```

# Esempio : la classe Student

```
class Student{
    //altri metodi

    public int hashCode(){
        int hcMatricola = matricola.hashCode();
        int hcNome = nome.hashCode();
        int hcDataNascita = data.Nascita.hashCode();
        int hcNumEsami = numEsami;
        final int HASH_MULTIPIER = 43;
        int h = hcMatricola;
        h = HASH_MULTIPIER * h + hcNome;
        h = HASH_MULTIPIER * h + hcDataNascita;
        h = HASH_MULTIPIER * h + hcNumEsami;
        return h;
    }

    String matricola;
    Name nome;
    Date dataNascita;
    int numEsami;
}
```

Attenzione!! Non è necessario considerare tutte le variabili d'istanza



# In generale . . .

- Usare un numero primo come moltiplicatore

`HASH_MULTIPLIER`

- Calcolare il codice hash di ogni campo
- Per gli interi usare il valore stesso

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

# In generale . . .

---

- Il metodo `hashCode` deve essere compatibile con il metodo `equals`  
`if x.equals(y) then x.hashCode() == y.hashCode()`
- E' necessario ridefinire `hashCode` quando si ridefinisce `equals`
  - Altrimenti uno dei due verrà ereditato e si darà origine ad una inconsistenza

# File Coin.java

```
01:  /**
02:      A coin with a monetary value.
03:  */
04:  public class Coin
05:  {
06:      /**
07:          Constructs a coin.
08:          @param aValue the monetary value of the coin.
09:          @param aName the name of the coin
10:      */
11:      public Coin(double aValue, String aName)
12:      {
13:          value = aValue;
14:          name = aName;
15:      }
16:
```

# File Coin.java

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

# 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:
```

# 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 hashCodeTester.java

```
01: import java.util.HashSet;
02: import java.util.Iterator;
03: import java.util.Set;
04:
05: /**
06:     A program to test hash codes of coins.
07: */
08: public class HashCodeTester
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:
```

# File hashCodeTester.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 hashCodeTester.java

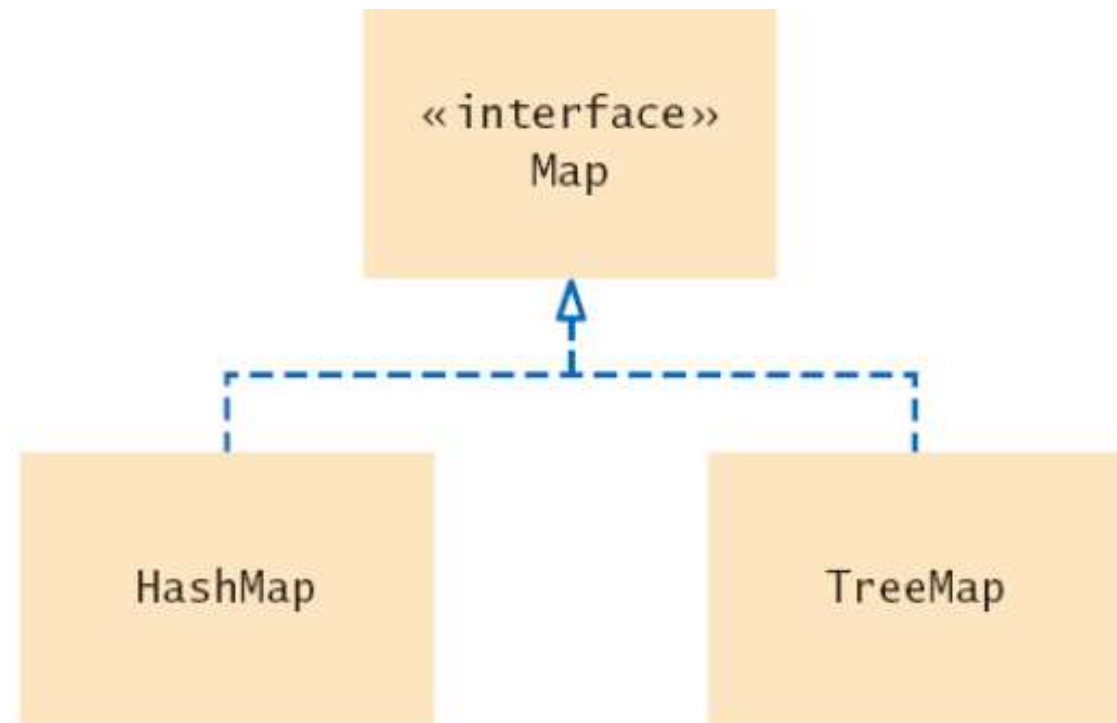
---

## ➤ Output

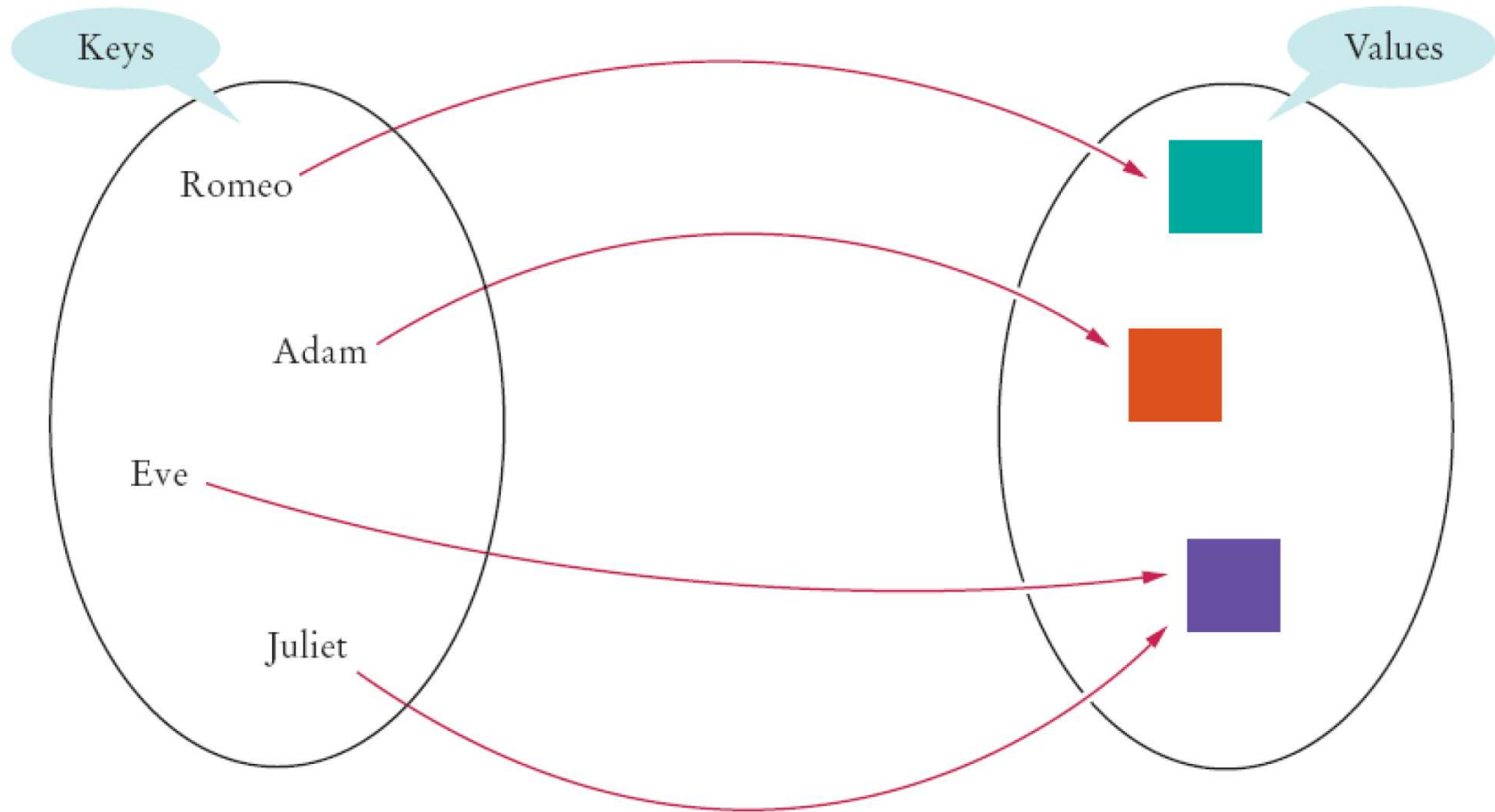
```
hash code of coin1=-1513525892  
hash code of coin2=-1513525892  
hash code of coin3=-1768365211  
Coin[value=0.25,name=quarter]  
Coin[value=0.05,name=nickel]
```

# Maps

- Una mappa memorizza coppie di oggetti (chiave valore)
  - Le chiavi identificano univocamente un valore
- Come per gli insiemi, i Set:
- **Map** interface
  - HashMap
  - TreeMap



# Un esempio di una Map



# HashMap

---

```
//Creating a HashMap  
Map<String, Color> favoriteColors  
    = new HashMap<String, Color>();
```

```
//Adding an association  
favoriteColors.put("Juliet", Color.PINK);
```

```
//Changing an existing association  
favoriteColor.put("Juliet", Color.RED);
```

# HashMap

---

```
//Getting the value associated with a key  
Color julietsFavoriteColor  
    = favoriteColors.get("Juliet");
```

```
//Removing a key and its associated value  
favoriteColors.remove("Juliet");
```

# Stampare tutte le coppie

---

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

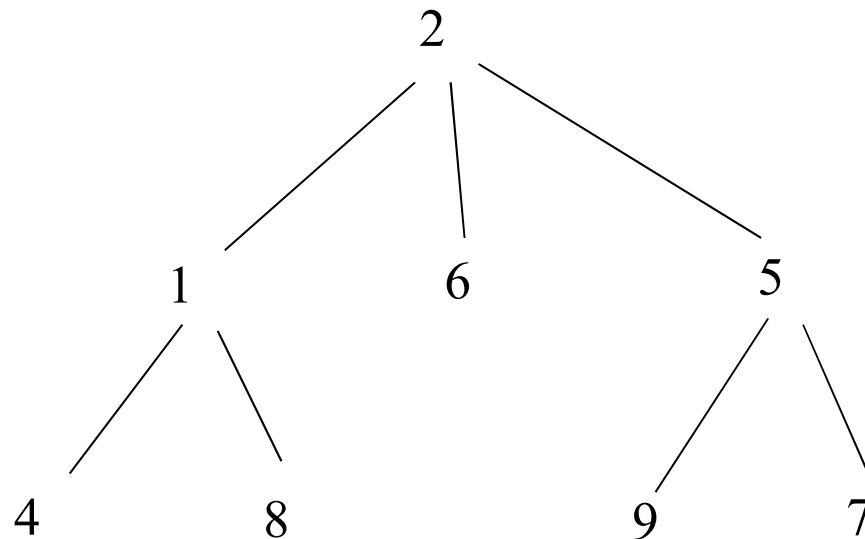
# Hash Maps

---

- Le chiavi sono hashed
  - L'hash code è calcolato sull'oggetto chiave
  - La coppia (chiave, valore) è memorizzata all'interno di una hash table nella posizione calcolata sulla base dall'hash code della chiave

# TreeSet e TreeMap

- È possibile usare una struttura dati ad albero per implementare Set e Map
- Un albero è un particolare tipo di grafo





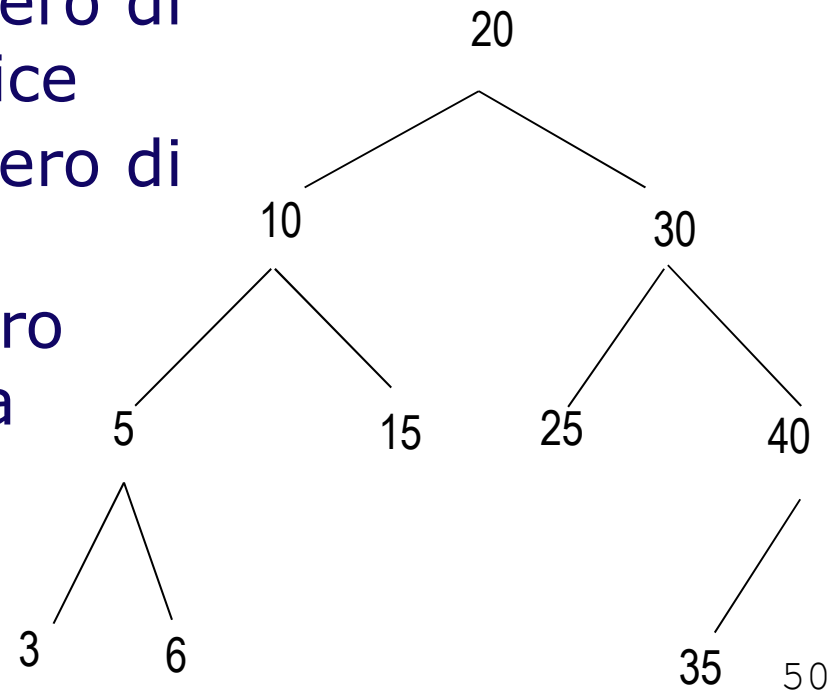
# Alberio binario

---

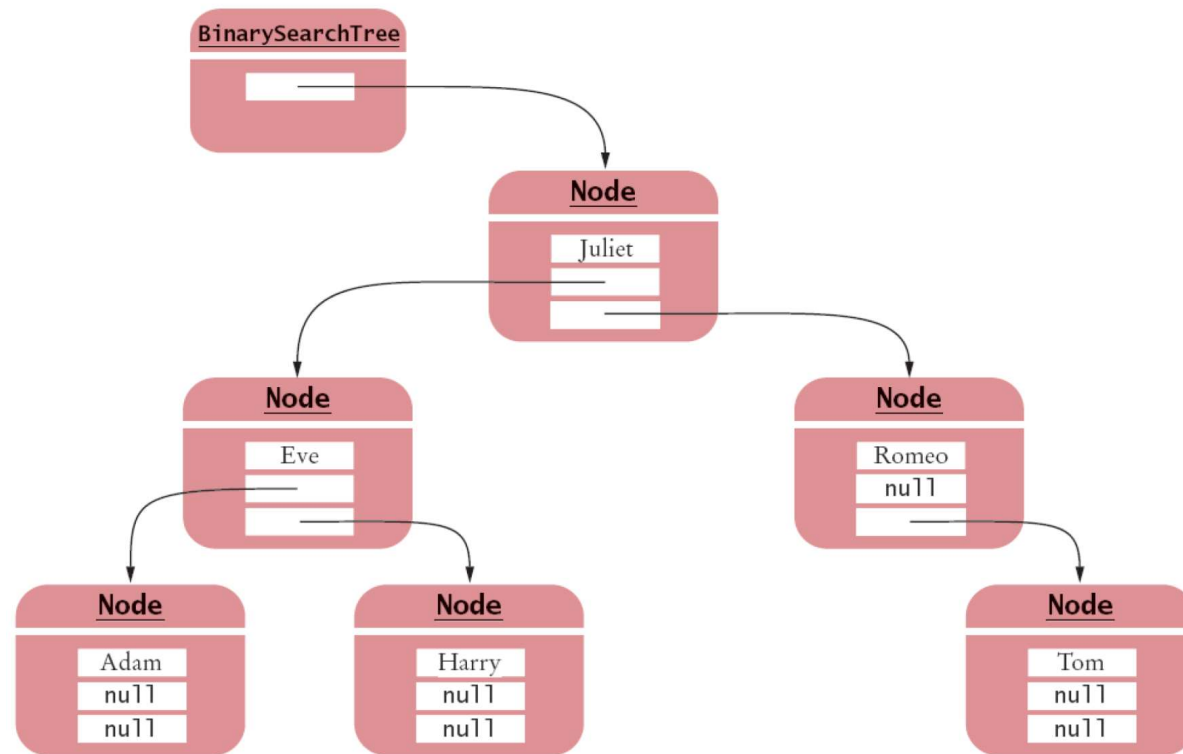
- Tipo particolare di albero:
- Ogni nodo può avere al più due figli
  - figlio sinistro e figlio destro
  - sottoalbero sinistro e sottoalbero destro
- Definizione ricorsiva:
  - un albero binario è vuoto
  - oppure è una terna  $(s, r, d)$ , dove  $r$  è un nodo (la radice),  $s$  e  $d$  sono alberi binari

# Alberi binari di ricerca

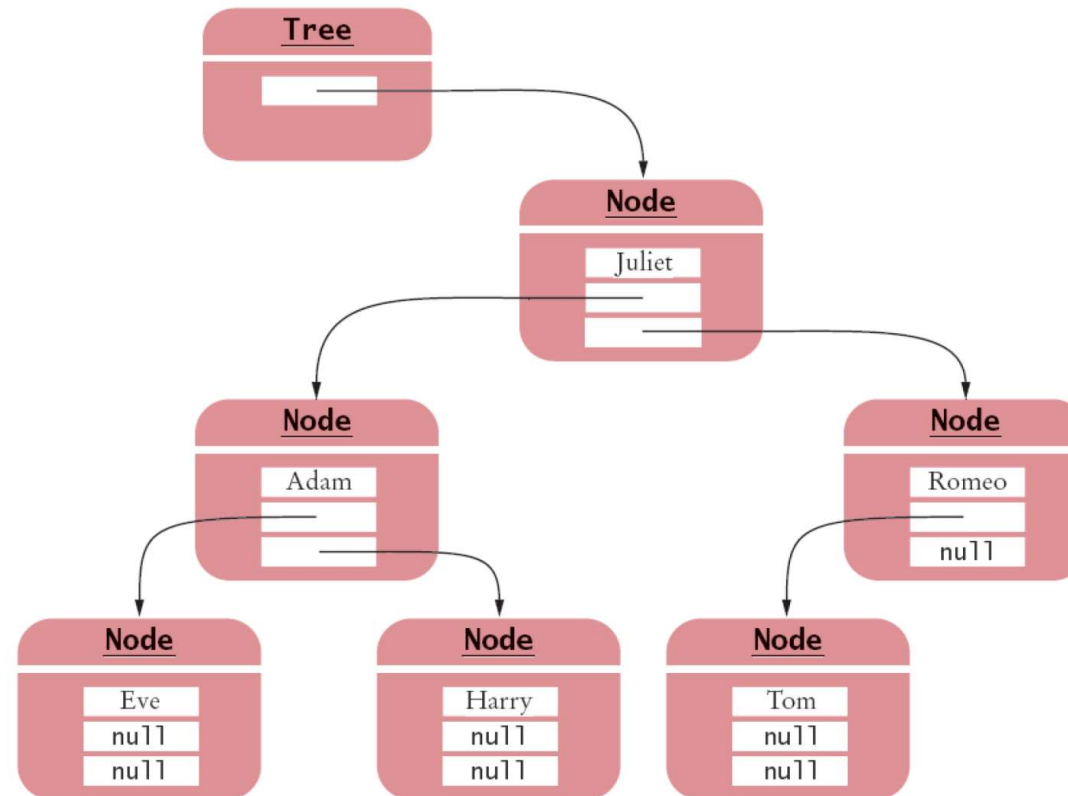
- Utilizzati nell'implementazione di HashTree e HashMap
- Molto efficienti in fase di aggiunta e ricerca di un elemento
- Se l'albero non è vuoto
  - ogni elemento del sottoalbero di sinistra precede ( $<$ ) la radice
  - ogni elemento del sottoalbero di destra segue ( $>$ ) la radice
  - i sottoalberi sinistro e destro sono alberi binari di ricerca



# Un esempio



# Un albero biario (non di ricerca)



# Realizzazione

---

- Un riferimento alla radice (Nodo)
- Ogni nodo contiene:
  - due riferimenti a Nodo
    - i figli destro e sinistro
  - un campo informazione
  - il campo informazione e di tipo `Comparable`
    - Deve essere necessario confrontare gli oggetti

# Realizzazione

```
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;
    }
}
```

# Inserimento

---

- Per ogni riferimento a nodo `non-null`, si analizza il valore di `data`
  - Se il valore di `data` è maggiore di quello da inserire, il processo continua con l'albero di sinistra
  - Se il valore di `data` è minore, il processo continua con l'albero destro
- Quando si raggiunge un riferimento nullo, si aggiunge il nuovo nodo

# Esempio

```
BinarySearchTree tree = new BinarySearchTree();
```

① `tree.add("Juliet");`

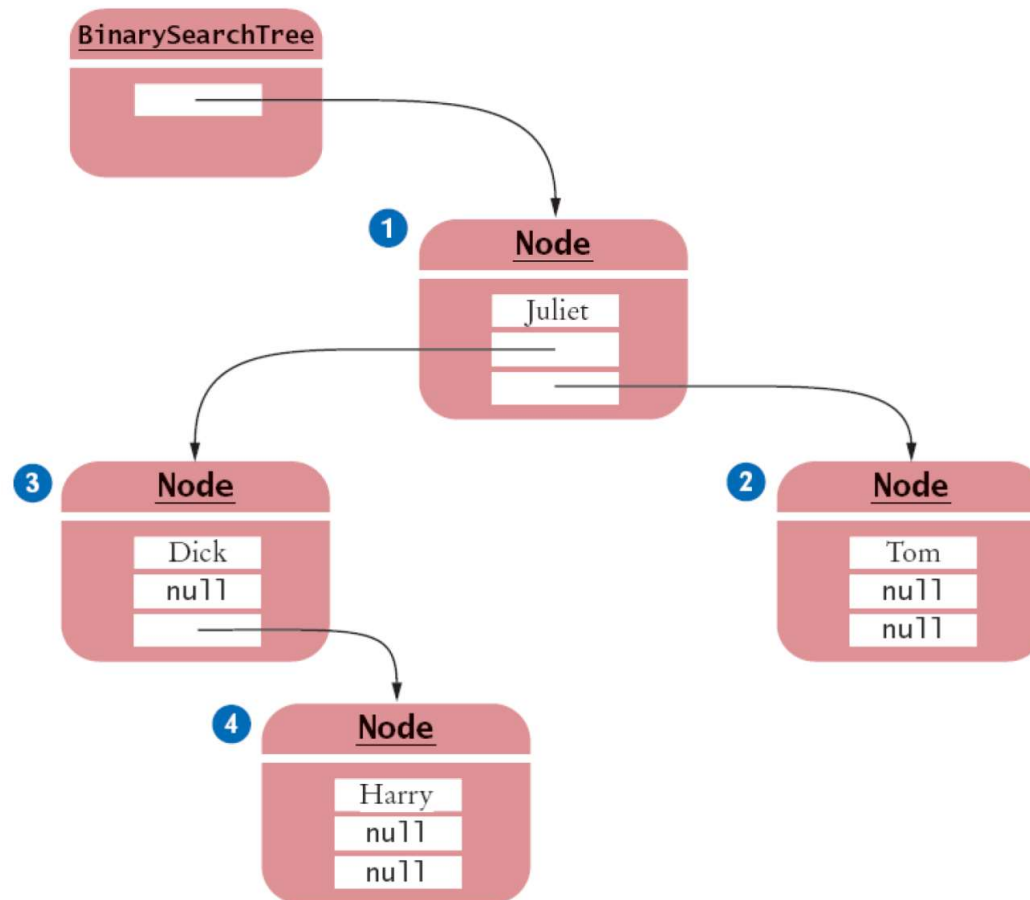
② `tree.add("Tom");`

③ `tree.add("Dick");`

④ `tree.add("Harry");`

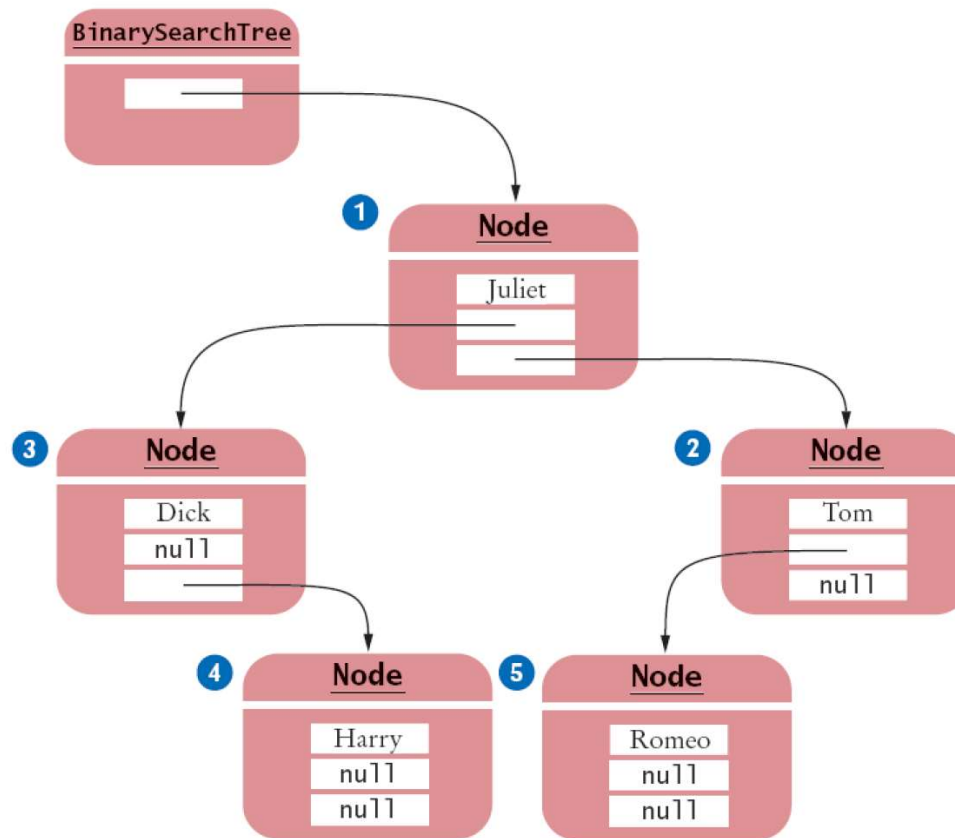


# Esempio



# Esempio

Tree: Add Romeo



# add

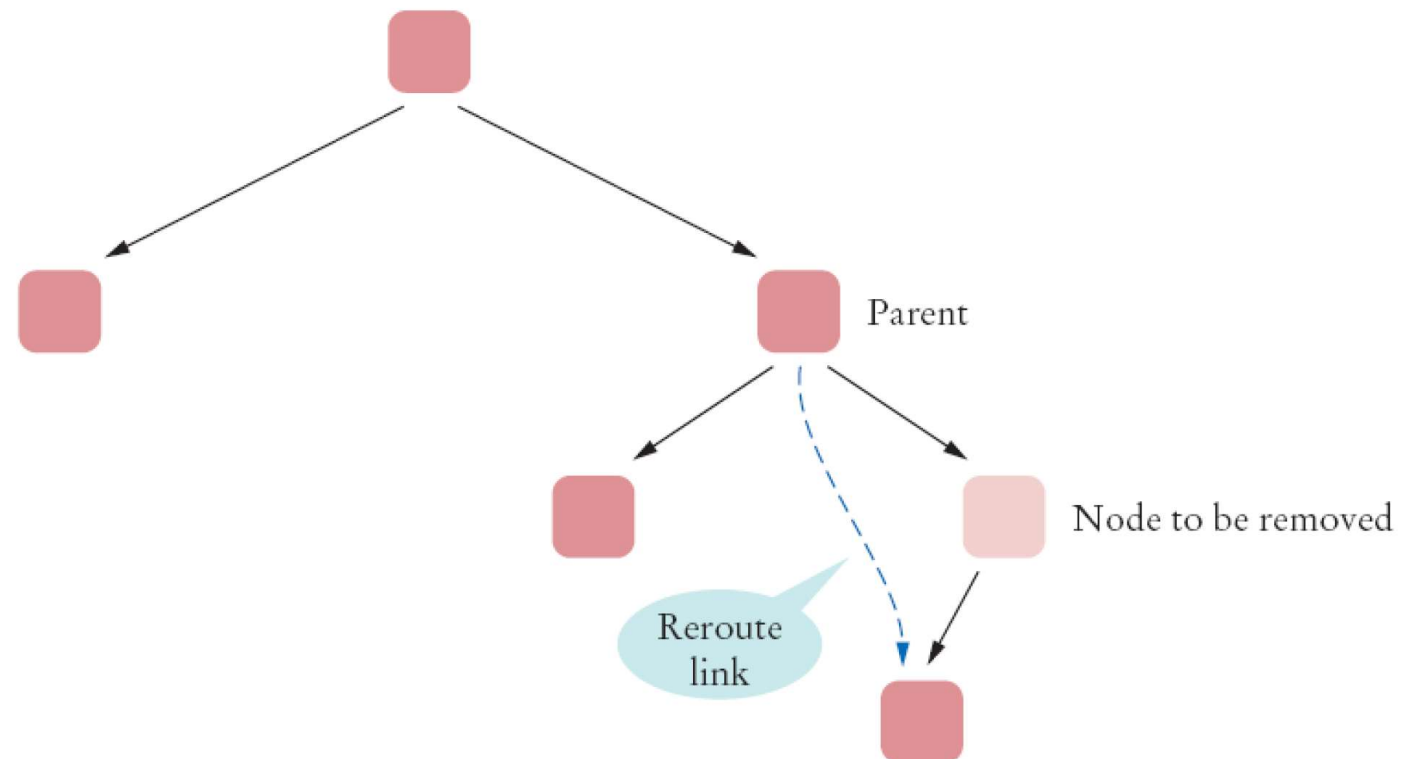
```
public class BinarySearchTree
{
    . . .
    public void add(Comparable obj)
    {
        Node newNode = new Node();
        newNode.data = obj;
        newNode.left = null;
        newNode.right = null;
        if (root == null) root = newNode;
        else root.addNode(newNode);
    }
    . .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

# add

```
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);  
        }  
    }  
    . . .  
}
```

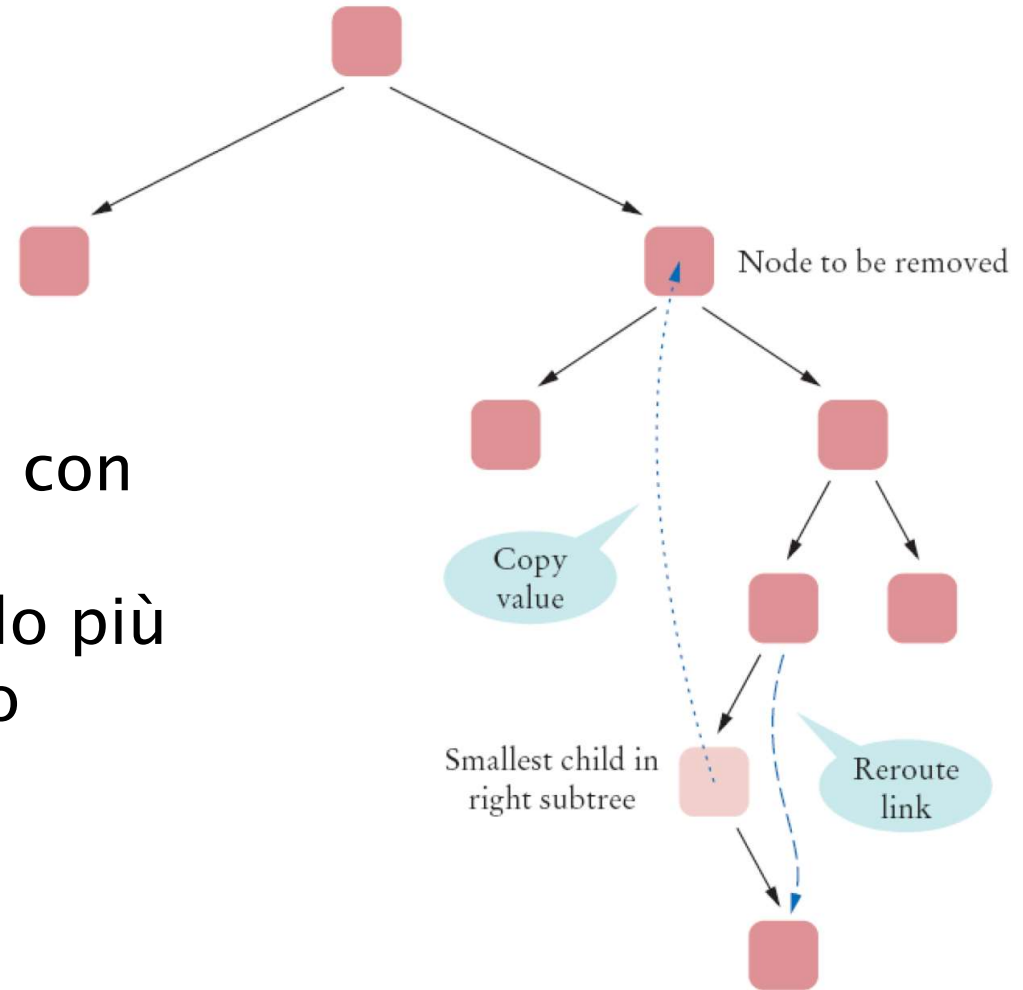
# Rimozione di un nodo

- È facile rimuovere un nodo foglia ...
- Se si rimuove un nodo con un solo figlio, il figlio rimpiazza il nodo rimosso



## ... e con due figli

- Se si rimuove un nodo con due figli, il nodo è rimpiazzato con il nodo più piccolo del sottoalbero destro



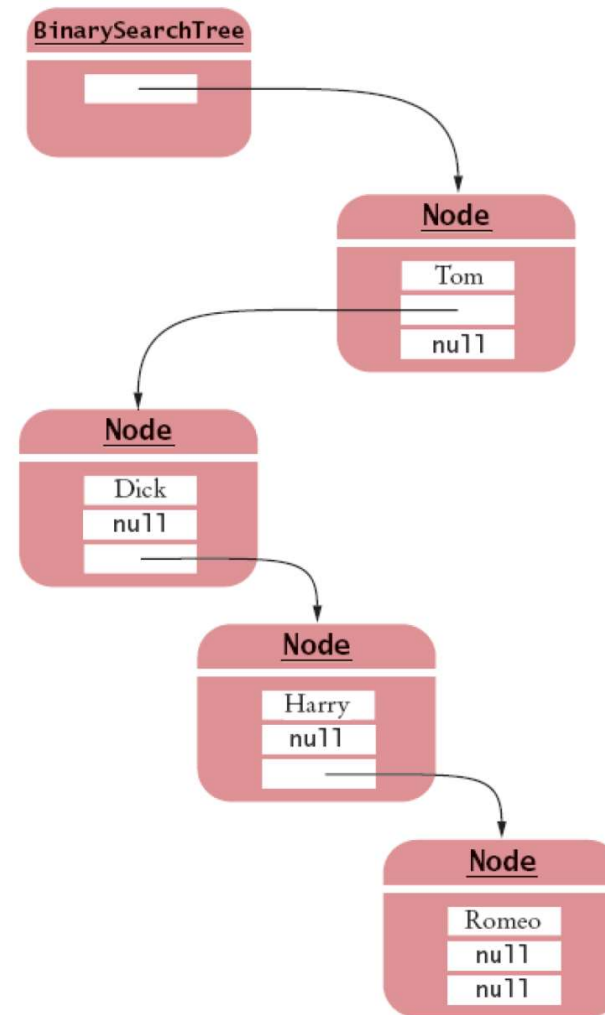
# Bilanciamento

---

- Albero bilanciato: ogni nodo ha un numero di discendenti a destra paragonabile a quello dei discendenti a sinistra
- Per un albero bilanciato l'aggiunta di un elemento ha complessità  $O(\log(n))$
- L'operazione diventa meno efficiente per alberi non bilanciati
  - Fino a diventare lineare . . .

# Un albero non bilanciato

Se si deve  
aggiungere il  
nodo "Terry"?





# File BinarySearchTree.java

---

```
001: /**
002:     This class implements a binary search tree whose
003:     nodes hold objects that implement the Comparable
004:     interface.
005: */
006: public class BinarySearchTree
007: {
008:     /**
009:         Constructs an empty tree.
010:     */
011:     public BinarySearchTree()
012:     {
013:         root = null;
014:     }
015:
```

# File BinarySearchTree.java

```
016:    /**
017:        Inserts a new node into the tree.
018:        @param obj the object to insert
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:
```

# File BinarySearchTree.java

```
030:      /**
031:          Tries to find an object in the tree.
032:          @param obj the object to find
033:          @return true if the object is in the tree
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:
```

# File BinarySearchTree.java

```
048:      /**
049:          Tries to remove an object from the tree. Does
050:          nothing if the object is not in the tree.
051:          @param obj the object to remove
052:      */
053:      public void remove(Comparable obj)
054:      {
055:          // Find node to be removed
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:              {
```

# 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 contains obj
075:
076: // If one of the children is empty, use the other
077:
078: if (toBeRemoved.left == null
    || toBeRemoved.right == null)
079: {
080:     Node newChild;
081:     if (toBeRemoved.left == null)
082:         newChild = toBeRemoved.right;
```

# File BinarySearchTree.java

```
083:         else
084:             newChild = toBeRemoved.left;
085:
086:         if (parent == null) // Found 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:     // Neither subtree is empty
096:
097:     // Find smallest element of the right subtree
098:
```

# 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 contains smallest child in right subtree
108:
109:         // Move contents, unlink child
110:
111:         toBeRemoved.data = smallest.data;
112:         smallestParent.left = smallest.right;
113:     }
114:
```

# File BinarySearchTree.java

```
115:    /**
116:        Prints the contents of the tree in sorted order.
117:    */
118:    public void print()
119:    {
120:        if (root != null)
121:            root.printNodes();
122:    }
123:
124:    private Node root;
125:
126:    /**
127:        A node of a tree stores a data item and references
128:        of the child nodes to the left and to the right.
129:    */
130:    private class Node
131:    {
```



# File BinarySearchTree.java

```
132:      /**
133:          Inserts a new node as descendant of this node
134:          @param newNode the node to insert
135:      */
136:      public void addNode(Node newNode)
137:      {
138:          if (newNode.data.compareTo(data) < 0)
139:          {
140:              if (left == null) left = newNode;
141:              else left.addNode(newNode);
142:          }
143:          else
144:          {
145:              if (right == null) right = newNode;
146:              else right.addNode(newNode);
147:          }
148:      }
149:
```

# File BinarySearchTree.java

```
150:      /**
151:          Prints this node and all of its descendants
152:          in sorted order.
153:      */
154:      public void printNodes()
155:      {
156:          if (left != null)
157:              left.printNodes();
158:          System.out.println(data);
159:          if (right != null)
160:              right.printNodes();
161:      }
162:
163:      public Comparable data;
164:      public Node left;
165:      public Node right;
166:  }
167: }
```

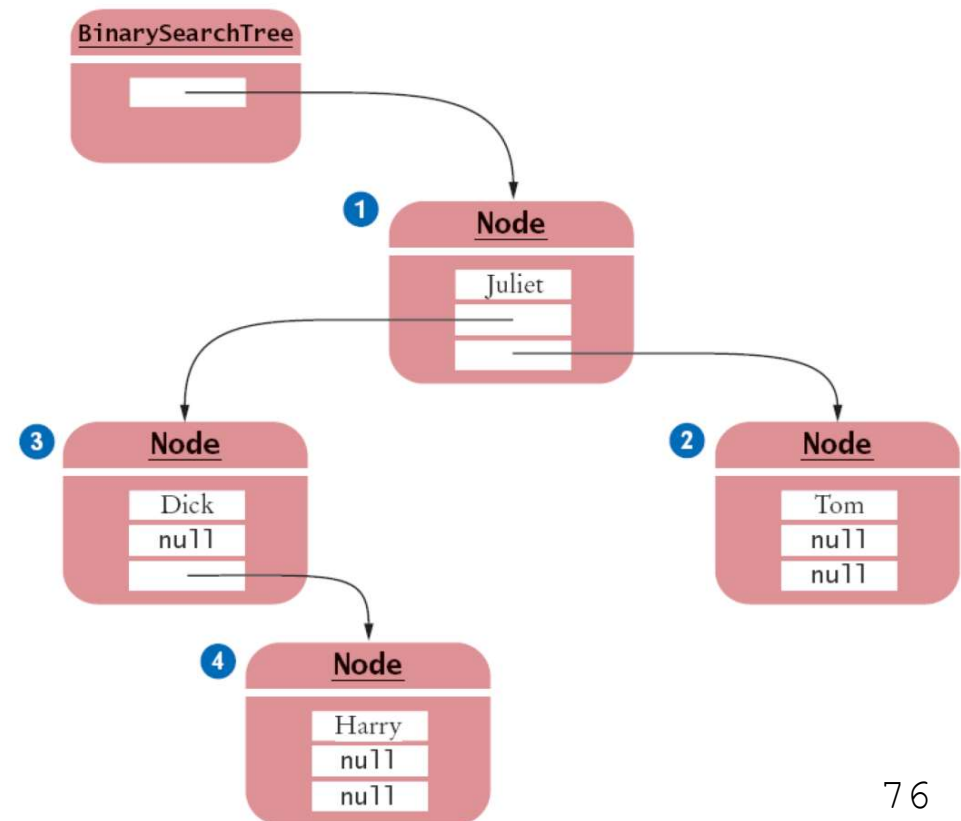
# Visita di un albero

---

- Stampa ordinata
  - Si stampa l'albero di sinistra
  - Si stampa il nodo
  - Si stampa l'albero di destra

# Esempio

1. Print the left subtree of Juliet; that is, Dick and descendants
2. Print Juliet
3. Print the right subtree of Juliet; that is, Tom and descendants



# Esempio

---

➤ Output:

```
Dick  
Harry  
Juliet  
Tom
```

# Metodi print e printNodes

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

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

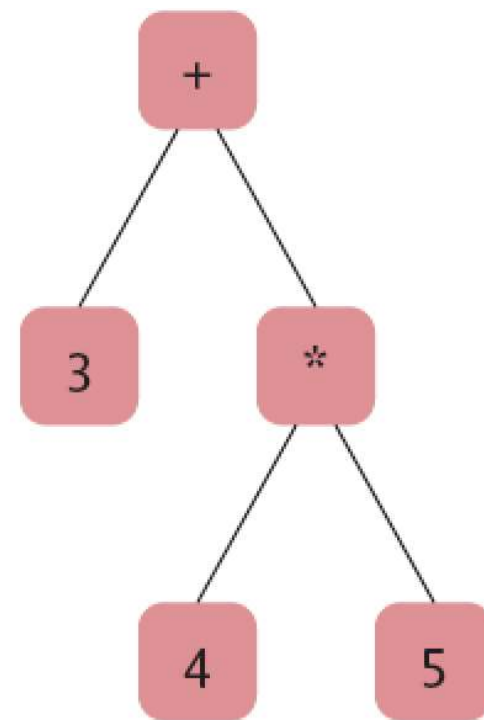
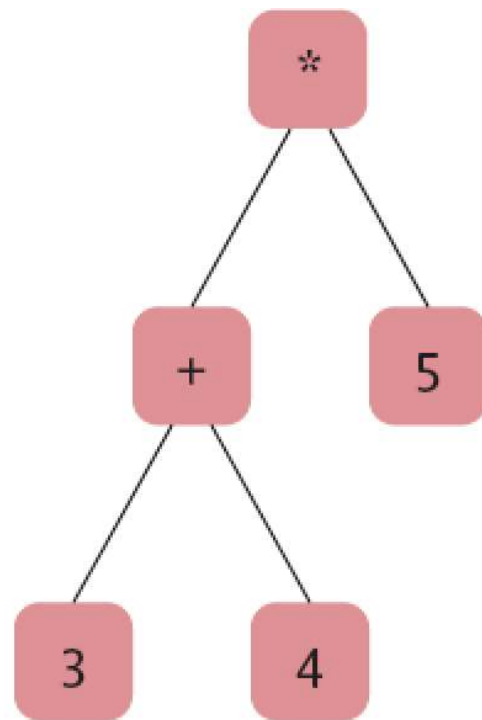
# Visite

---

- Tre schemi
  - Preorder traversal
    - root
    - left subtree
    - right subtree
  - Inorder traversal
    - left subtree
    - root
    - right subtree
  - Postorder traversal
    - left subtree
    - right subtree
    - root

# Visite

- Un'espressione aritmetica può essere modellata attraverso un albero binario
- La visita postorder di un albero che descrive una espressione aritmetica definisce l'ordine di esecuzione delle operazioni per valutare l'espressione con una calcolatrice stack-based

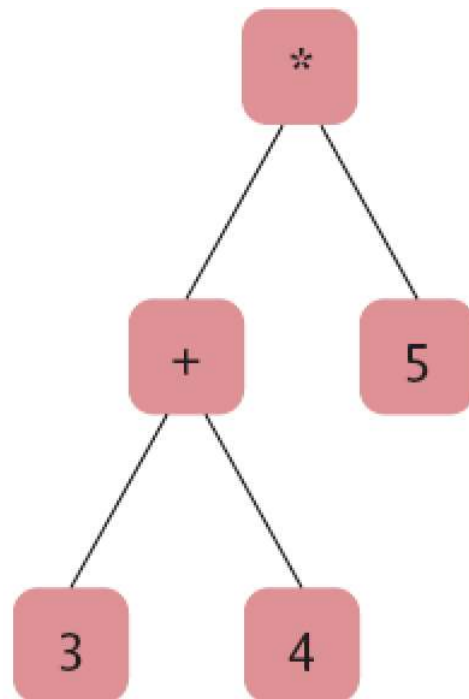




# Esempio

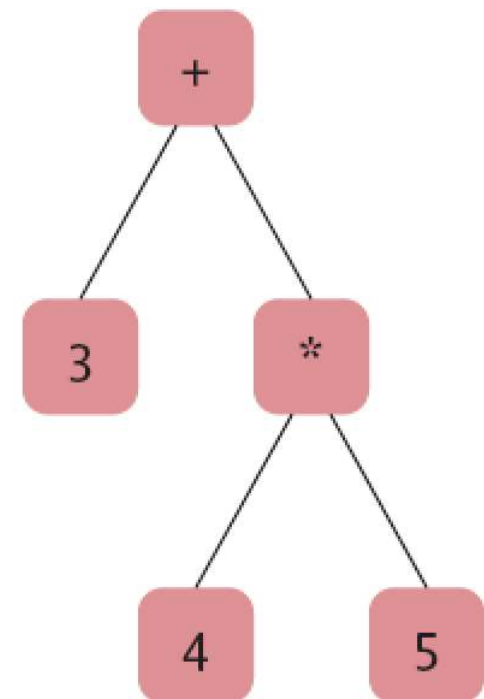
$((3 + 4) * 5)$

3 4 + 5 \*



$(3 + 4 * 5)$

3 4 5 \* +



# Per usare un TreeSet

---

- Gli oggetti devono implementare l'interfaccia `Comparable`
  - O deve essere definito un opportuno oggetto di tipo `Comparator`

# Per usare una TreeMap

---

- La chiave deve implementare l'interfaccia `Comparable`
  - O deve essere definito un oggetto `Comparator` per le chiavi
- Nessun vincolo sui valori

# File TreeSetTester.java

```
01: import java.util.Comparator;
02: import java.util.Iterator;
03: import java.util.Set;
04: import java.util.TreeSet;
05:
06: /**
07:     A program to test hash codes of coins.
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:
```

# File TreeSetTester.java

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

# File TreeSetTester.java

---

```
34:
35:     for (Coin c : coins)
36:         System.out.println(c);
37:     }
38: }
```

# File TreeSetTester.java

---

## ➤ Output:

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

# Esercizi

---

- Scrivere un programma che restituisce l'elenco in ordine alfabetico delle parole contenute in un testo, ognuna accompagnata dal numero di occorrenze
- Scrivere un programma che analizza un insieme di file java, i cui nomi sono contenuti in un file denominato "system.list", e restituisce l'elenco di tutte le classi contenute, ognuna accompagnata con l'elenco dei nomi dei metodi che la classe definisce