

21 - Alberi e Ricorsione

Programmazione e analisi di dati
Modulo A: Programmazione in Java

Paolo Milazzo

Dipartimento di Informatica, Università di Pisa
<http://www.di.unipi.it/~milazzo>
milazzo@di.unipi.it

Corso di Laurea Magistrale in Informatica Umanistica
A.A. 2014/2015

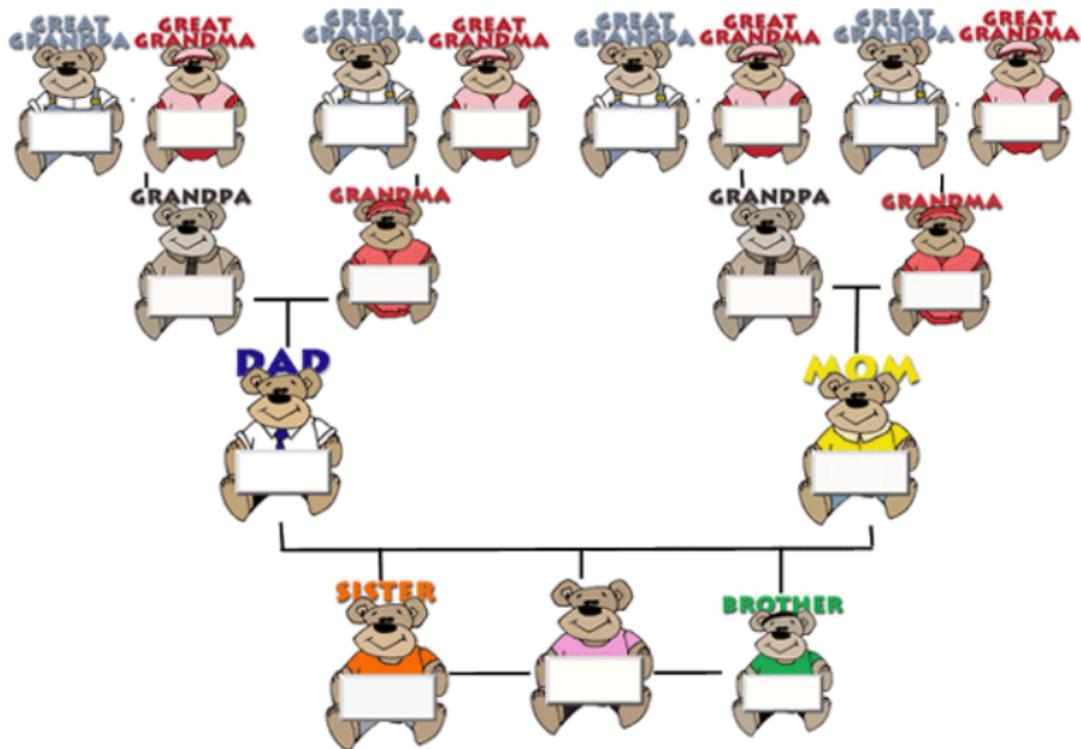
Altre strutture dati...

In certe situazioni gli array, i vettori e le altre strutture dati fornite dalla Libreria di Java non sono semplici da usare...

Esempio: **albero genealogico**

- Supponiamo di voler scrivere un programma che gestisce un albero genealogico
- Abbiamo un **individuo**, di cui dobbiamo indicare i genitori, i nonni, i bisnonni, ecc...
- Possiamo assumere di rappresentare l'**individuo** tramite un **oggetto**
- Il metodo **setGenitori** consente di specificare i genitori dell'individuo
- I genitori saranno a loro volta individui... di cui si potranno specificare i genitori, ecc...

Albero genealogico (1)



Albero genealogico (2)

Come deve essere fatta la **classe Individuo**?

- Deve prevedere una stringa per il nome dell'individuo
- Deve prevedere riferimenti ai due individui genitori
 - ▶ ossia, due variabili di tipo **Indiviuo**

Quindi: la classe **Individuo** conterrà riferimenti a oggetti della stessa classe!

Albero genealogico (3)

```
public class Individuo {  
  
    // nome dell'individuo  
    private String nome;  
  
    // riferimento al padre (di tipo Individuo)  
    private Individuo padre;  
  
    // riferimento alla madre (di tipo Individuo)  
    private Individuo madre;  
  
    // costruttore: inizializza solo il nome  
    public Individuo(String nome) {  
        // padre e madre inizializzati di default a null  
        this.nome = nome;  
    }  
}
```

(segue)

Albero genealogico (4)

(segue Individuo)

```
// restituisce il nome
public String getNome() { return nome; }

// restituisce l'individuo padre
public Individuo getPadre() { return padre; }

// imposta il padre (se non ancora impostato)
public void setPadre(Individuo padre) {
    if (this.padre == null) this.padre = padre;
}

// restituisce l'individuo madre
public Individuo getMadre() { return madre; }

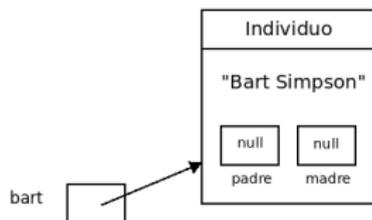
// imposta la madre (se non ancora impostata)
public void setMadre(Individuo madre) {
    if (this.madre==null) this.madre = madre;
}
}
```

Albero genealogico (5)

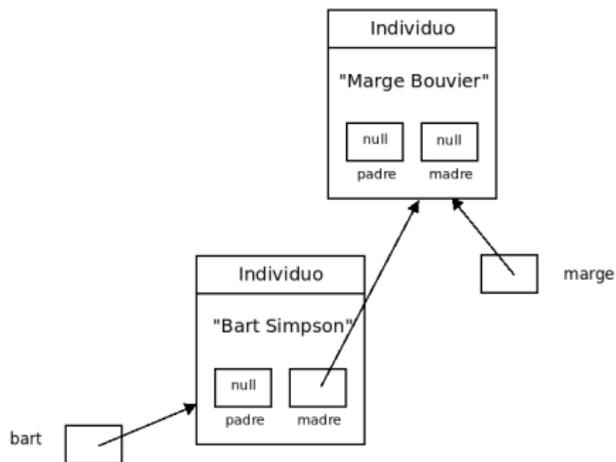
Come si usa? Vediamo un main...

```
public class UsaAlberoGenealogico {  
    public static void main(String[] args) {  
        Individuo bart = new Individuo("Bart Simpson");  
  
        Individuo marge = new Individuo("Marge Bouvier");  
        bart.setMadre(marge);  
  
        Individuo homer = new Individuo("Homer Simpson");  
        bart.setPadre(homer);  
  
        Individuo nonno1 = new Individuo("Abraham Simpson");  
        homer.setPadre(nonno1); // nota: metodo invocato su homer!  
  
        Individuo nonna1 = new Individuo("Mona Simpson");  
        homer.setMadre(nonna1); // nota: metodo invocato su homer!  
  
        Individuo nonno2 = new Individuo("Clancy Bouvier");  
        marge.setPadre(nonno2); // nota: metodo invocato su marge!  
  
        Individuo nonna2 = new Individuo("Jacqueline Bouvier");  
        marge.setMadre(nonna2); // nota: metodo invocato su marge!  
    }  
}
```

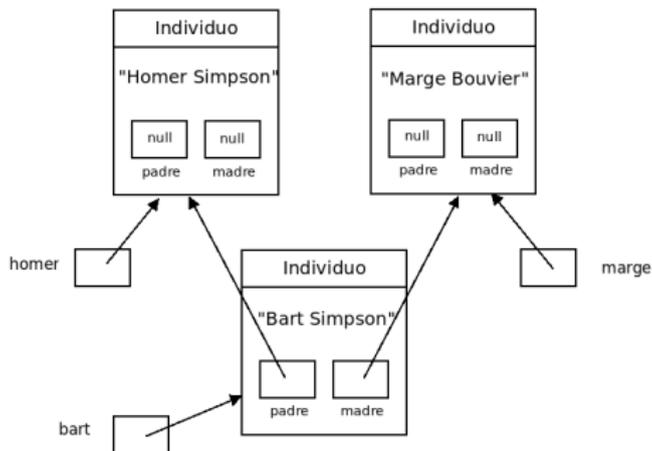
Albero genealogico (6a)



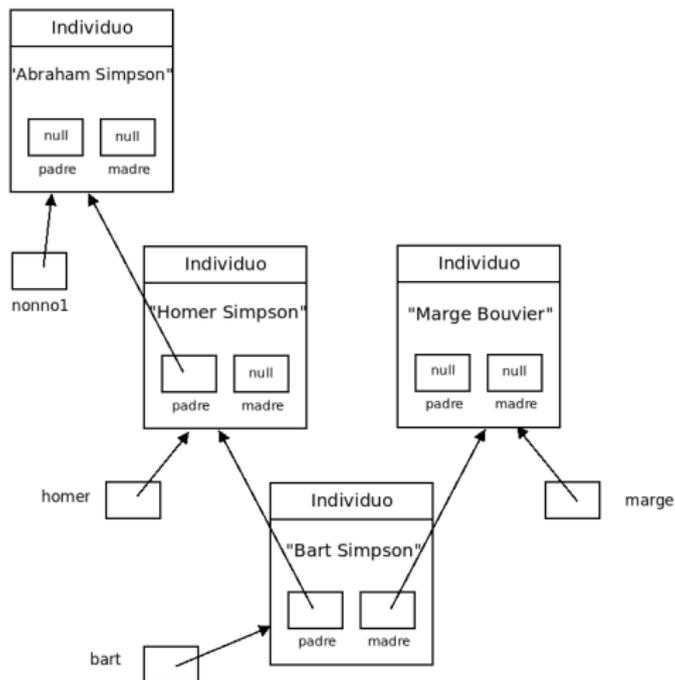
Albero genealogico (6b)



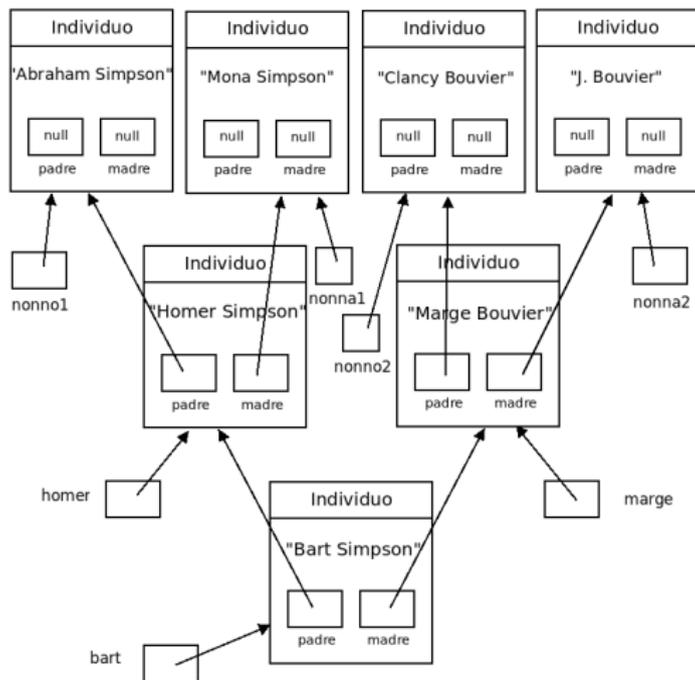
Albero genealogico (6c)



Albero genealogico (6d)



Albero genealogico (6e)



Albero genealogico (7)

E se volessimo stampare l'albero genealogico (tramite un nuovo metodo `stampaAlberoGenealogico()`)?

Ad esempio, cercando di ottenere questo risultato:

```
Bart Simpson
--Homer Simpson
----Abraham Simpson
----Mona Simpson
--Marge Simpson
----Clancy Bouvier
----Jacqueline Bouvier
```

Albero genealogico (8)

Per poter **stampare l'albero genealogico** di un individuo, il metodo `stampaAlberoGenealogico()` dovrebbe:

- stampare il nome dell'individuo su cui è chiamato
- **stampare l'albero genealogico** del padre
- **stampare l'albero genealogico** della madre

Ma....

- anche il padre e la madre **sono individui...**
- la funzionalità di stampa dell'albero genealogico di un individuo **la stiamo definendo ora...**
- questo è un **cane che si morde la coda...**

Albero genealogico (9)

Tutto sotto controllo!

Il metodo `stampaAlberoGenealogico()` che stiamo definendo è **ricorsivo**

- al suo interno **richiama se stesso!**

```
//nuovo metodo della classe Individuo  
  
public void stampaAlberoGenealogico() {  
    System.out.println(nome);  
    if (padre!=null) padre.stampaAlberoGenealogico();  
    if (madre!=null) madre.stampaAlberoGenealogico();  
}
```

```
//utilizzo del metodo (aggiunto in fondo al main)  
  
bart.stampaAlberoGenealogico();
```

Albero genealogico (10)

Attenzione: il controllo `padre!=null` (analogo per madre) serve per due motivi:

- se il padre (o la madre) è `null` non abbiamo niente da stampare per quanto lo riguarda
- se la variabile `padre` è `null`, il comando `padre.stampaAlberoGenealogico()` da un **errore** a tempo di esecuzione
 - ▶ eccezione `NullPointerException`

Albero genealogico (11)

Risultato:

```
Bart Simpson  
Homer Simpson  
Abraham Simpson  
Mona Simpson  
Marge Bouvier  
Clancy Bouvier  
Jacqueline Bouvier
```

Mancano i trattini prima dei nomi degli avi!

Albero genealogico (11)

Proviamo a modificare il metodo...

```
//nuovo metodo della classe Individuo

public void stampaAlberoGenealogico() {
    System.out.println(nome);
    if (padre!=null) {
        System.out.print("--");
        padre.stampaAlberoGenealogico();
    }
    if (madre!=null) {
        System.out.print("--");
        madre.stampaAlberoGenealogico();
    }
}
```

Albero genealogico (12)

Risultato:

```
Bart Simpson
--Homer Simpson
--Abraham Simpson
--Mona Simpson
--Marge Bouvier
--Clancy Bouvier
--Jacqueline Bouvier
```

Non ci siamo ancora...

- Il numero di trattini che devono essere stampati in ogni riga dipende da “quanto vecchio” è l’individuo
- ossia, da quanto è lontano dall’individuo **radice** dell’albero (bart)

Soluzione: aggiungere un contatore da passare da un individuo ai suoi genitori incrementandolo ogni volta

Albero genealogico (13)

Nuovo tentativo...

- Ora il metodo ricorsivo prevede anche un parametro intero (il contatore)

```
//nuovo metodo della classe Individuo

public void stampaAlberoGenealogico(int contatore) {
    // stampa tanti trattini quanti indicati da contatore
    for (int i=0; i<contatore; i++)
        System.out.println("--");
    System.out.println(nome);

    // chiamate ricorsive con contatore aumentato
    if (padre!=null) padre.stampaAlberoGenealogico(contatore+1);
    if (madre!=null) madre.stampaAlberoGenealogico(contatore+1);
}
```

Modifichiamo anche la chiamata nel main

```
// all'inizio il contatore e' 0 (non servono trattini per bart)
bart.stampaAlberoGenealogico(0);
```

Albero genealogico (14)

Risultato:

```
Bart Simpson
--Homer Simpson
----Abraham Simpson
----Mona Simpson
--Marge Bouvier
----Clancy Bouvier
----Jacqueline Bouvier
```

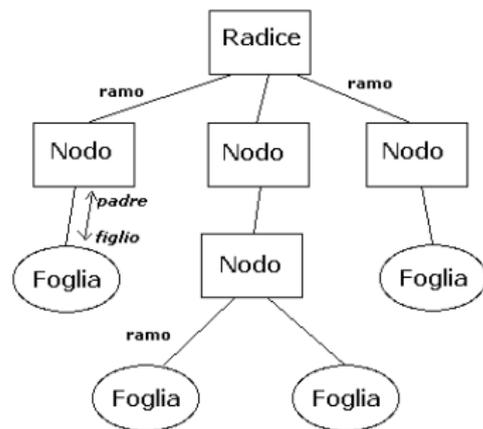
Ci siamo!

La struttura dati Albero (1)

La struttura dati che abbiamo usato per realizzare l'albero genealogico prende il nome di **Albero**

Un albero:

- E' costituito da **nodi** (in Java sono oggetti) tra i quali esiste una **relazione padre-figlio**
- Ha un nodo **radice** (senza padri)
- Ha nodi **foglia** (senza figli)

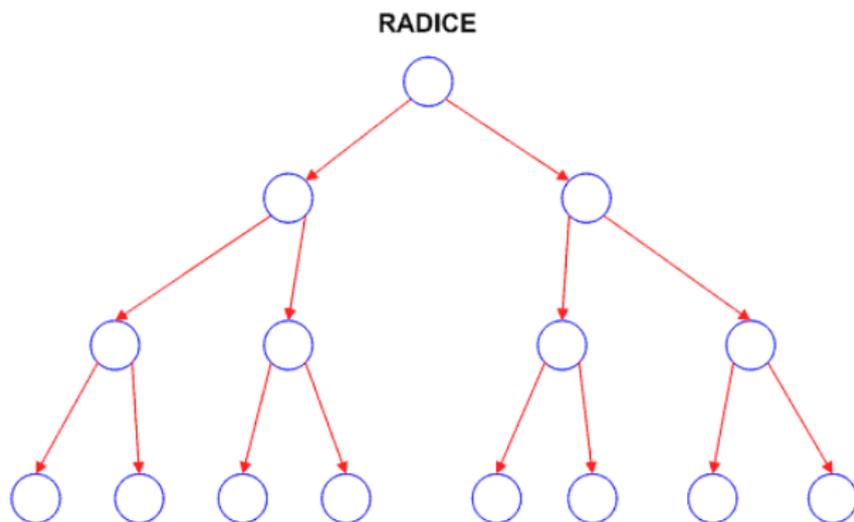


Un **sottoalbero** è una porzione di un albero che consiste di un nodo intermedio e tutti i suoi discendenti.

La struttura dati Albero (2)

Il **numero di figli** di ogni nodo di un albero può essere fissato o variabile

Quando il numero dei figli di ogni nodo è **fissato a 2**, l'albero si dice **binario** (come nell'esempio dell'albero genealogico)



Questo è un albero binario

La struttura dati Albero (3)

Quando il numero dei figli di ogni nodo è **fissato a 1**, l'albero in realtà descrive una **lista concatenata**

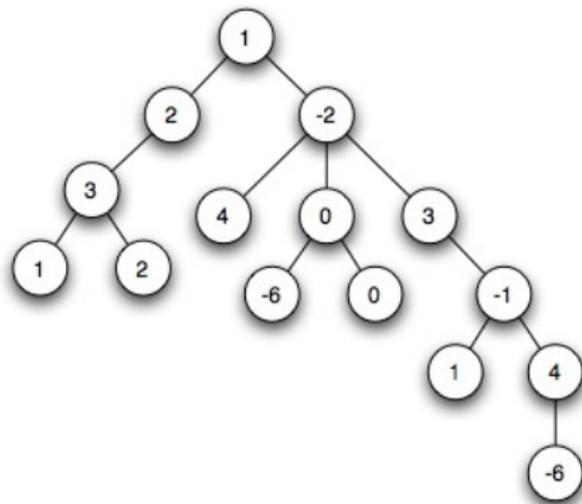


Una lista concatenata è una struttura dati che in Java si usa raramente

- le sue funzionalità (aggiungere e togliere elementi dalla sequenza) sono simili a quelle di `Vector`, `ArrayList` e altre strutture dati presenti nella Libreria Standard

La struttura dati Albero (4)

Quando il numero dei figli di ogni nodo **può variare**, in Java di solito i riferimenti ai figli si memorizzano tramite vettori



La struttura dati Albero (5)

Esempio: I generi letterari dei libri di una biblioteca

- Ogni generi (es. prosa, poesia,)
- ... ha tanti sotto-generi (es. romanzi, racconti,)
- ... che prevedono sotto-sotto-generi (es. romanzo storico, romanzo di avventura, ...)

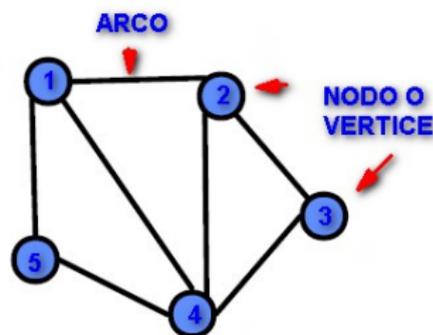
```
public class GenereLetterario {  
  
    private String nomeGenere;  
    private Vector<GenereLetterario> sottogeneri;  
  
    //... costruttori e metodi  
}
```

La struttura dati Grafo

Generalizzando un po' la definizione di albero, otteniamo un **grafo**

Un grafo:

- E' costituito da **nodi** (in Java sono oggetti)
- I nodi sono collegati da **archi** (in Java sono riferimenti, o oggetti aggiuntivi)



Molti programmi richiedono l'uso di grafi (es. gestione reti ferroviarie e stradali, social network, World Wide Web,)

Gli algoritmi che utilizzano grafi possono essere piuttosto **complicati**

- si va ben oltre gli scopi di questo corso...

La ricorsione (1)

Un programma è detto **ricorsivo** quando risolve un problema di una certa dimensione riutilizzando se stesso su un problema di dimensione minore

Nell'esempio dell'albero genealogico:

- il metodo che stampa l'**albero** (di un individuo) riutilizza se stesso per stampare un **sottoalbero** (di un genitore dell'individuo)

L'**approccio ricorsivo** si contrappone all'**approccio iterativo** in cui i problemi sono risolti mediante cicli

L'approccio ricorsivo può essere molto efficace (consente di scrivere alcuni programmi in poche righe) ma:

- può essere difficile da imparare ad utilizzare propriamente
- se usato impropriamente può causare problemi (es. mancata terminazione del programma)

La ricorsione (2)

Confrontiamo l'approccio iterativo e l'approccio ricorsivo.

Esempio (classico): Il calcolo del fattoriale $n! = n \cdot n - 1 \cdot \dots \cdot 2 \cdot 1$

```
// soluzione iterativa
private int fattorialeIterativo(int n) {
    int ris = 1;
    for (int i=1; i<=n; i++)
        ris *= i;
    return ris;
}
```

```
// soluzione ricorsiva
private int fattorialeRicorsivo(int n) {
    if (n==0) return 1;
    else return n*fattorialeRicorsivo(n-1);
}
```

La ricorsione (3)

L'approccio ricorsivo è abbastanza comune in **matematica**:

- definiamo la **funzione matematica** *fattoriale*(n)

$$fattoriale(n) = \begin{cases} 1 & \text{se } n = 0; \\ n \cdot fattoriale(n - 1) & \text{altrimenti} \end{cases}$$

La ricorsione (4)

Come tutte le chiamate di metodi, le chiamate ricorsive sono gestite tramite **record di attivazione** (rivedere lezioni precedenti)

La chiamata ricorsiva:

- **sospende** il metodo in corso di esecuzione
- attiva il metodo chiamato (una nuova istanza dello stesso metodo)
- quando al termine del metodo chiamato il controllo ritorna al chiamante

La ricorsione (5)

Esempio di metodo ricorsivo:

```
public void esempioRicorsione(int i) {  
    if (i==0) System.out.println("ECCO 0");  
    else {  
        System.out.println("PRIMA " + i);  
        esempioRicorsione(i-1);  
        System.out.println("DOPO " + i);  
    }  
}
```

Chiamando il metodo come segue:

```
esempioRicorsivo(3);
```

Il risultato sarà:

```
PRIMA 3  
PRIMA 2  
PRIMA 1  
ECCO 0  
DOPO 1  
DOPO 2  
DOPO 3
```

La ricorsione (6)

Un metodo ricorsivo deve prevedere un numero di **casi** a seconda della dimensione del problema da affrontare

Dobbiamo avere:

- uno o più **casi base** che risolvono il problema senza bisogno di chiamate ricorsive
- uno o più **casi ricorsivi** che hanno bisogno di effettuare chiamate ricorsive per risolvere sottoproblemi

```
private int fattorialeRicorsivo(int n) {  
    if (n==0) return 1; // caso base  
    else return n*fattorialeRicorsivo(n-1); // caso ricorsivo  
}
```

La ricorsione (7)

La condizione necessaria affinché la ricorsione possa funzionare è che le chiamate ricorsive prima o poi **raggiungano un caso base**

- altrimenti il programma **non terminerebbe mai...** eseguirebbe chiamate ricorsive all'infinito

Vediamo un esempio di programma ricorsivo mal posto.

- che succede se si invoca il metodo con parametro dispari?

```
private int fattorialeDifettoso(int n) {  
    if (n==0) return 1; // caso base  
    else return n*fattorialeRicorsivo(n-2); // caso ricorsivo  
}
```

La ricorsione (8)

Rivediamo l'esempio della stampa dell'albero genealogico

```
//nuovo metodo della classe Individuo

public void stampaAlberoGenealogico(int contatore) {
    // stampa tanti trattini quanti indicati da contatore
    for (int i=0; i<contatore; i++)
        System.out.println("--");
    System.out.println(nome);

    // chiamate ricorsive con contatore aumentato
    if (padre!=null) padre.stampaAlberoGenealogico(contatore+1);
    if (madre!=null) madre.stampaAlberoGenealogico(contatore+1);
}
```

In questo esempio:

- Abbiamo **ricorsione multipla** (2 chiamate ricorsive)
- Il caso base lo si ha quando padre e madre sono entrambi null (si stampa solo il nome)
- Il sottoproblema non è identificato dal **parametro** del metodo, ma dall'**oggetto** su cui il metodo è invocato
 - ▶ quando è chiamato su bart, radice dell'albero intero...
 - ▶ ... si richiama ricorsivamente su homer e marge, radici di sottoalberi

Esempio: la ricerca dicotomica (o binaria) (1)

Vediamo un esempio complesso di programma ricorsivo:

- La **ricerca dicotomica** (o **ricerca binaria**)

Supponiamo di avere un **array ordinato** contenente interi

- vogliamo sapere se un certo numero **è contenuto** nell'array

Esempio: la ricerca dicotomica (o binaria) (2)

Prima soluzione (iterativa)

```
public class Ricerca {  
  
    public static void main(String[] args) {  
        // array ordinato di lunghezza 12  
        int[] a = { 2, 3, 6, 8, 9, 10, 14, 15, 21, 22, 24, 30, 41 };  
  
        boolean trovato = ricerca(a, 19); // restituisce false  
    }  
  
    // restituisce true se x e' presente in a  
    private boolean ricerca(int[] a, int x) {  
        boolean trovato = false;  
        for (int y : a) {  
            if (x==y) trovato=true;  
        }  
        return trovato;  
    }  
}
```

Esempio: la ricerca dicotomica (o binaria) (3)

Ragioniamo un attimo:

- l'algoritmo che abbiamo implementato va a guardare **tutti i valori** dell'array
 - ▶ se l'array è grande, questo può richiedere **tempo**
- non abbiamo sfruttato il fatto che gli elementi **sono ordinati**

Idea: come si cerca di solito una parola in un vocabolario?

Esempio: la ricerca dicotomica (o binaria) (4)

Altra soluzione:

- leggiamo il valore centrale dell'array e lo confrontiamo con il valore cercato
- il confronto ci dice se continuare la ricerca nella metà di destra o di sinistra dell'array
- ripetiamo la ricerca **ricorsivamente** nella metà selezionata

Quanti numeri dovremo leggere?

- Ad ogni passo ne buttiamo via la metà senza leggerli
- Alla fine avremo confrontato il numero cercato con un numero di valori dell'array pari al logaritmo della dimensione (quindi molti meno)
 - ▶ Nell'esempio i numeri erano 12, ma bastano 3 confronti...

Esempio: la ricerca dicotomica (o binaria) (5)

```
public class RicercaBinaria {

    public static void main(String[] args) {
        // array ordinato di lunghezza 12
        int[] a = { 2, 3, 6, 8, 9, 10, 14, 15, 21, 22, 24, 30, 41 };

        boolean trovato = ricercaBinaria(a, 0, a.length, 19); // restituisci

    }

    // restituisce true se x e' presente in a
    // sx e dx sono gli estremi destro e sinistro della porzione
    // di array che si considera
    public boolean ricercaBinaria (int[] a, int sx, int dx, int x) {
        boolean trovato;
        if (sx <= dx) {
            int centro = (sx+dx)/2; // calcola il centro
            if (x < a[centro])
                return ricercaBinaria(a, sx, centro-1, x);
            if (x > a[centro])
                return ricercaBinaria(a, centro+1, dx, x);
            if (x == a[centro])
                return true;
        }
        else
            return false;
    }
}
```