IEIM

Esercitazione IV

2014-2015

Alessandro A. Nacci
alessandro.nacci@polimi.it - alessandronacci.it


READY...
SET...
GO

RICORSIONE

Divide et impera

- Metodo di approccio ai problemi che consiste nel dividere il problema dato in problemi più semplici
- I risultati ottenuti risolvendo i problemi più semplici vengono combinati insieme per costituire la soluzione del problema originale
- Generalmente, quando la semplificazione del problema consiste essenzialmente nella semplificazione dei DATI da elaborare (ad es. la riduzione della dimensione del vettore da elaborare), si può pensare ad una soluzione ricorsiva

La ricorsione (1)

- Una funzione è detta **ricorsiva** se chiama se stessa
- Se due funzioni si chiamano l'un l'altra, sono dette **mutuamente ricorsive**
- La funzione ricorsiva sa risolvere direttamente solo casi particolari di un problema detti **casi di base**: se viene invocata passandole dei dati che costituiscono uno dei casi di base, allora restituisce un risultato
- Se invece viene chiamata passandole dei dati che NON costituiscono uno dei casi di base, allora chiama se stessa (passo ricorsivo) passando dei DATI semplificati/ridotti

La ricorsione (II)

- Ad ogni chiamata si semplificano/riducono i dati, così ad un certo punto si arriva ad uno dei casi di base
- Quando la funzione chiama se stessa, sospende la sua esecuzione per eseguire la nuova chiamata
- L'esecuzione riprende quando la chiamata interna a se stessa termina
- La sequenza di chiamate ricorsive termina quando quella più interna (annidata) incontra uno dei casi di base
- *Ogni chiamata alloca sullo stack (in stack frame diversi) nuove istanze dei parametri e delle variabili locali (non static)*

Esempio: il fattoriale

Funzione ricorsiva che calcola il fattoriale di un numero n

Premessa (definizione ricorsiva):

se $n \leq 1$ → $n! = 1$

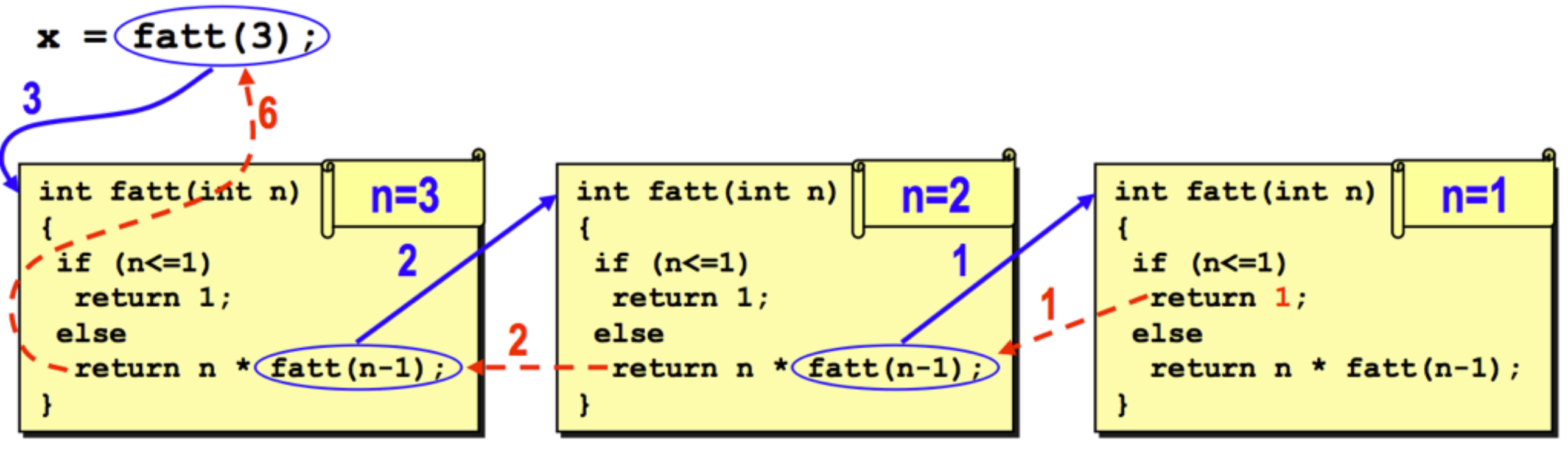
se $n > 1$ → $n! = n * (n-1)!$

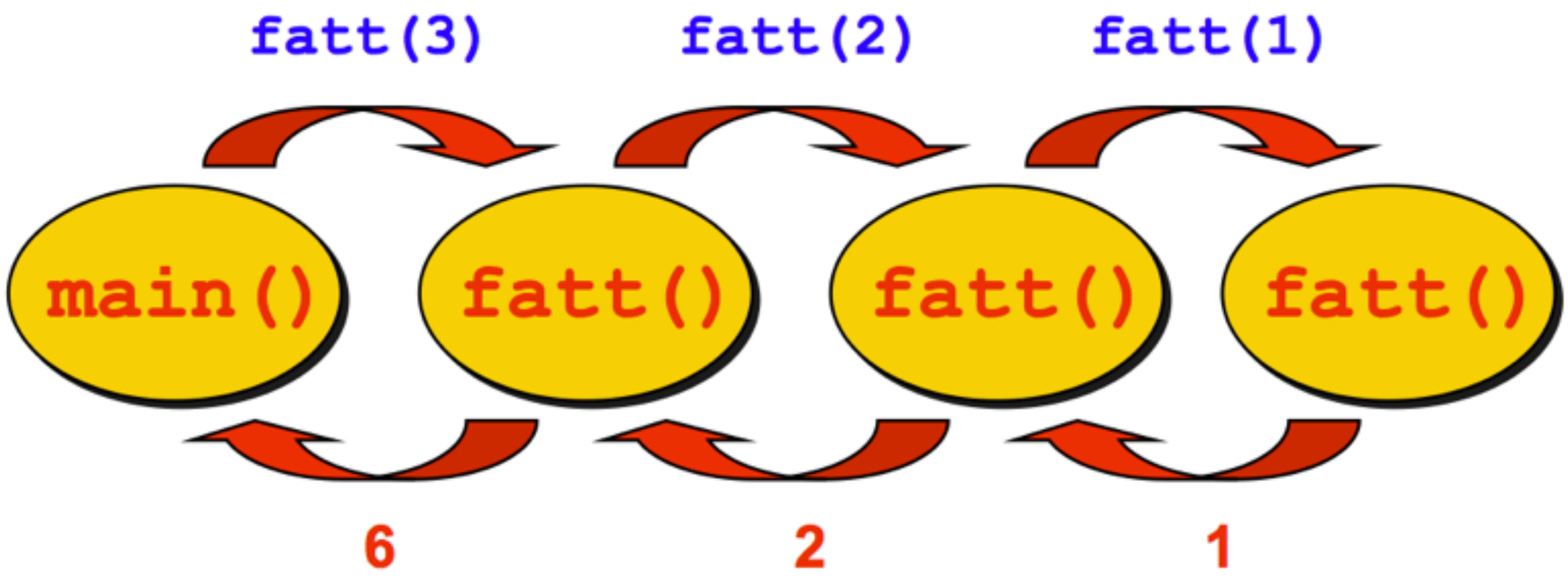
```
int fatt(int n)
{
   if (n<=1)
      return 1;
   else
      return n * fatt(n-1);
}
```

→ *Caso di base*

Semplificazione dei dati del problema

Esempio: il fattoriale - passo per passo





Ricorsione - Esercizio 1

- Scrivere una funziona ricorsiva che calcoli ricorsivamente la somma di tutti i numeri compresi tra 0 ed x
- Il prototipo della funzione è: int ric(int x)

```
int ric(int x) {



}
```

Ricorsione - Esercizio 1

- Scrivere una funziona ricorsiva che calcoli ricorsivamente la somma di tutti i numeri compresi tra 0 ed x
- Il prototipo della funzione è: int ric(int x)

```
int ric(int x) {
   if (x == 0)
       return 0;
   else
       return x + ric(x-1);
}
```

Ricorsione - Esercizio 11

- Scrivere le versioni ricorsiva ed iterativa di una funziona che calcoli il seguente valore

$$\sum_{i=1}^{n}\left(a-\frac{i}{a}\right)$$

```
double f(double a, int n)
{

}
```

Ricorsione - Esercizio 11

- Scrivere le versioni ricorsiva ed iterativa di una funziona che calcoli il seguente valore

$$\sum_{i=1}^{n}\left(a-\frac{i}{a}\right)$$

```
double f(double a, int n)
{

}
```

Ricorsione - Esercizio 11

- Scrivere le versioni ricorsiva ed iterativa di una funziona che calcoli il seguente valore

$$\sum_{i=1}^{n}\left(a-\frac{i}{a}\right)$$

```
double f(double a, int n)
{ if (n==1) return a - 1/a;
  else return a - n/a + f(a, n-1);
}
```

Ricorsione - Esercizio 11

- Scrivere le versioni ricorsiva ed iterativa di una funziona che calcoli il seguente valore

$$\sum_{i=1}^{n}\left(a-\frac{i}{a}\right)$$

```
double f(double a, int n)
{ if (n==1) return a - 1/a;
  else return a - n/a + f(a, n-1);
}
```

```
double f(double a, int n)
{ int i=1;
  double sum=0;
  while(i<=n)
       {sum = sum + a - i/a;
        i++;}
  return  sum;
}
```

Qualche considerazione

- L'apertura delle chiamate ricorsive semplifica il problema, ma non calcola ancora nulla
- Il valore restituito dalle funzioni viene utilizzato per calcolare il valore finale man mano che si chiudono le chiamate ricorsive: ogni chiamata genera valori intermedi a partire dalla fine
- Nella ricorsione vera e propria non c'è un mero passaggio di un risultato calcolato nella chiamata più interna a quelle più esterne, ossia le return non si limitano a passare indietro invariato un valore, ma c'è un'elaborazione intermedia

Quando utilizzare la ricorsione

- PRO
  Spesso la ricorsione permette di risolvere un problema anche molto complesso con poche linee di codice
- CONTRO
  La *ricorsione è poco efficiente* perché richiama molte volte una funzione e questo:
  - richiede tempo per la gestione dello stack (allocare e passare i parametri, salvare l'indirizzo di ritorno, e i valori di alcuni registri della CPU)
  - consuma molta memoria (alloca un nuovo stack frame ad ogni chiamata, definendo una nuova ulteriore istanza delle variabili locali non `static` e dei parametri ogni volta)

Quando utilizzare la ricorsione

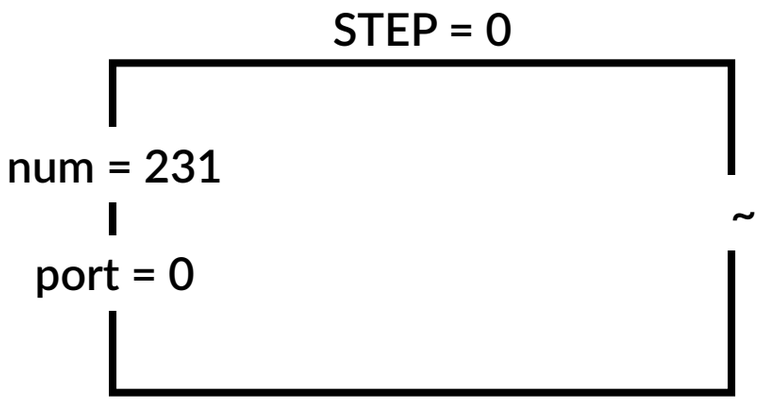
- CONSIDERAZIONE
  Qualsiasi problema ricorsivo può essere risolto in modo non ricorsivo (ossia iterativo), ma la soluzione iterativa potrebbe non essere facile da individuare oppure essere molto più complessa
- CONCLUSIONE
  Quando non ci sono particolari problemi di efficienza e/o memoria, l'approccio ricorsivo è in genere da preferire se:
  - è più intuitivo di quello iterativo
  - la soluzione iterativa non è evidente o agevole

Ricorsione - Esercizio III

- Analizzare il comportamento della seguente funzione ricorsiva mostrando l'andamento delle variabili, considerando i seguenti input:
  - *funzione(231,0)*
- Dire quale funzione espleta

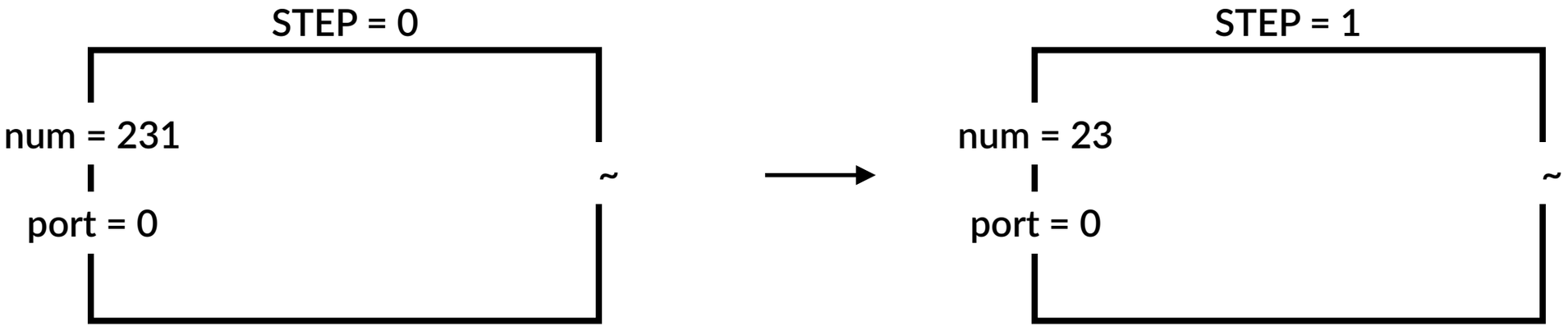
```
int mia_funzione(int num, int part) {
     if (num == 0)
          return part;
     else {
          return mia_funzione(num/10, part*10 + num%10);
     }
}
```

Ricorsione - Esercizio III - *Soluzione*



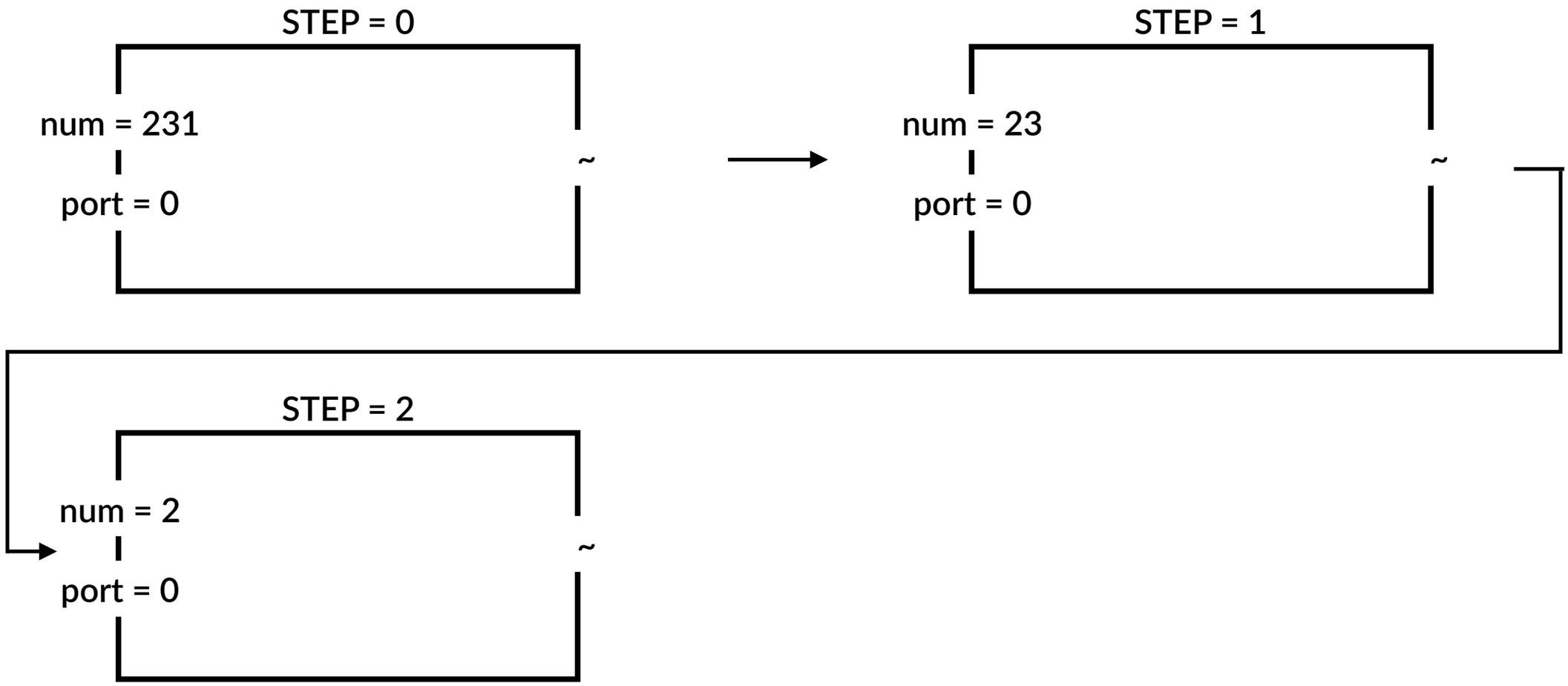

```
int funzione (int num, int port)
{
  if (num == 0)
    return port;
  else
    return funzione (num/10, port*10 + num%10);
}
```

Ricorsione - Esercizio III - *Soluzione*



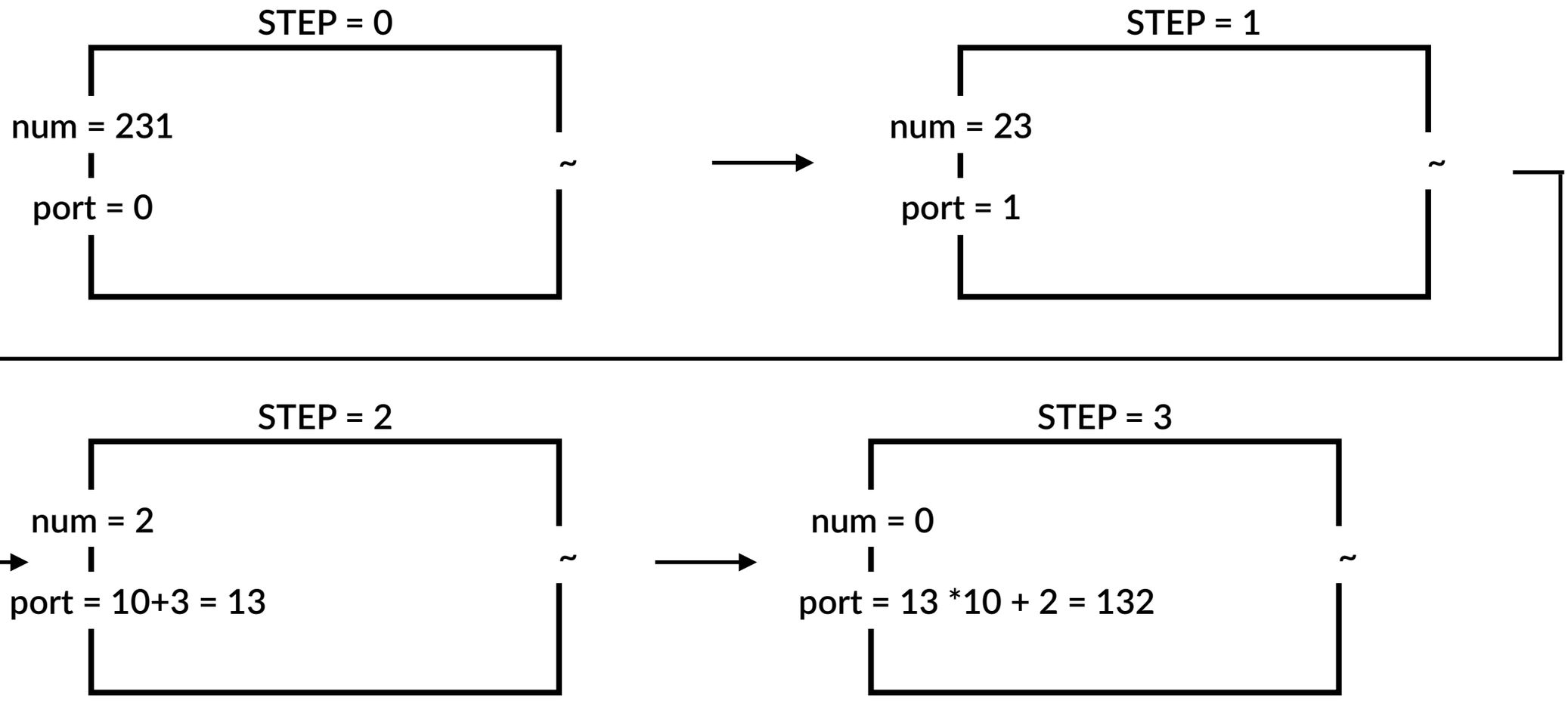

```
int funzione (int num, int port)
{
  if (num == 0)
    return port;
  else
    return funzione (num/10, port*10 + num%10);
}
```

Ricorsione - Esercizio III - *Soluzione*



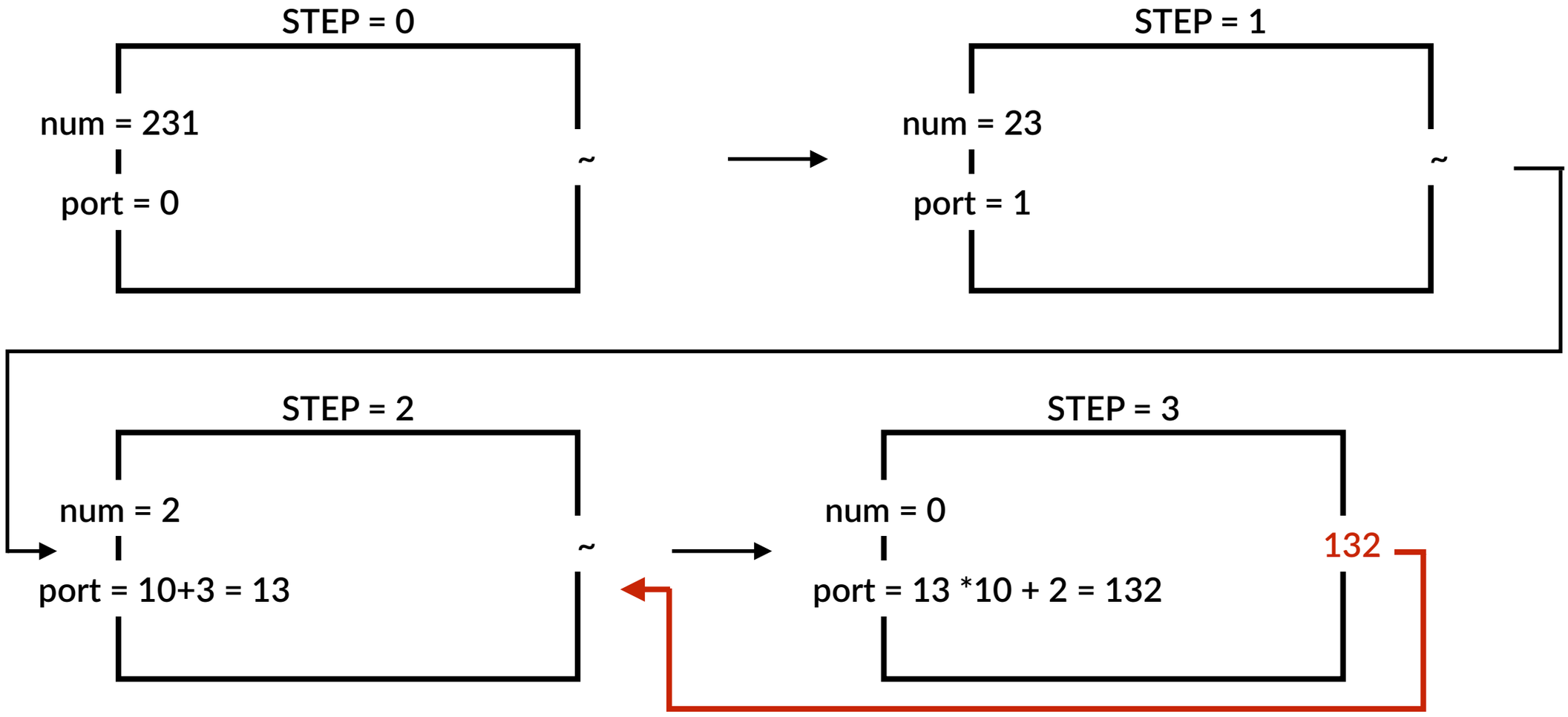

```
int funzione (int num, int port)
{
  if (num == 0)
    return port;
  else
    return funzione (num/10, port*10 + num%10);
}
```

Ricorsione - Esercizio III - *Soluzione*



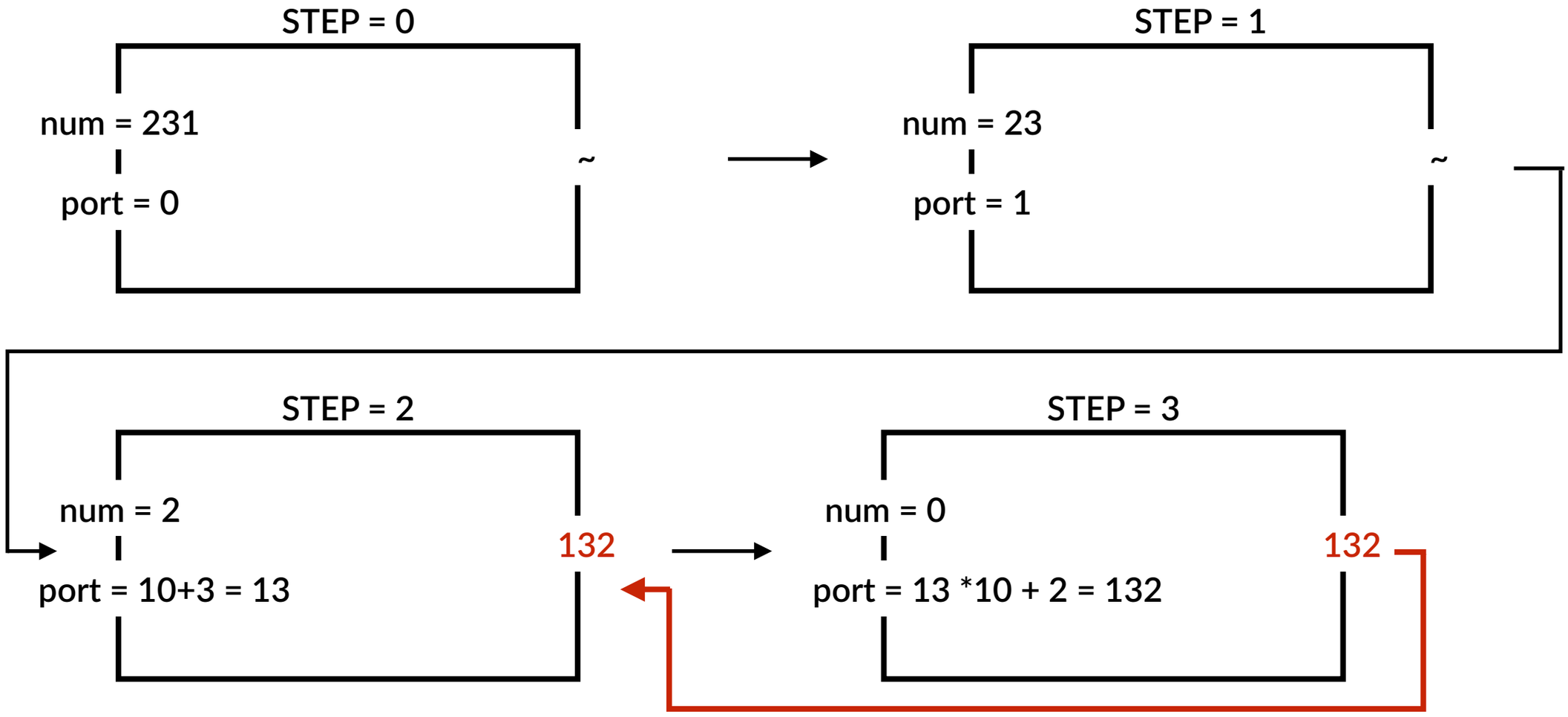

```
int funzione (int num, int port)
{
  if (num == 0)
    return port;
  else
    return funzione (num/10, port*10 + num%10);
}
```

Ricorsione - Esercizio III - *Soluzione*



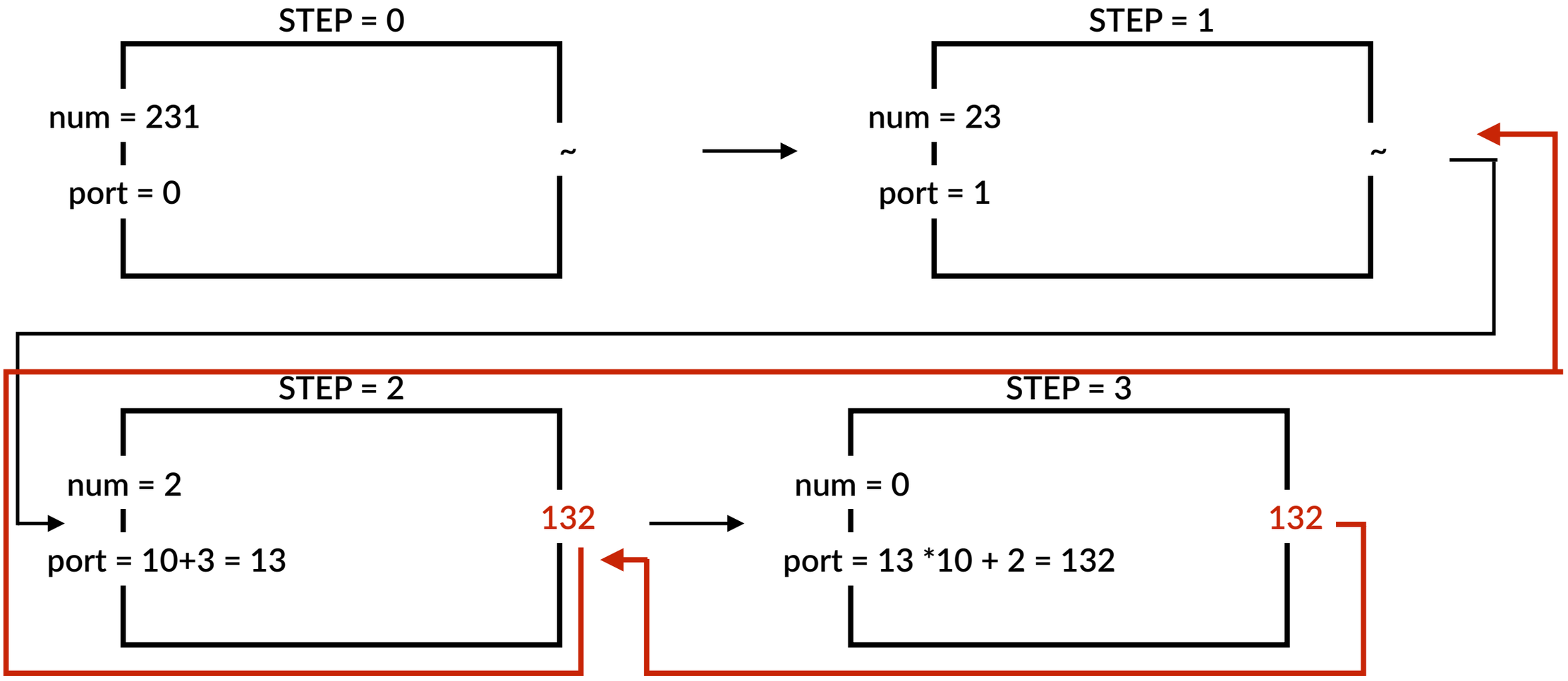

```
int funzione (int num, int port)
{
  if (num == 0)
    return port;
  else
    return funzione (num/10, port*10 + num%10);
}
```

Ricorsione - Esercizio III - *Soluzione*



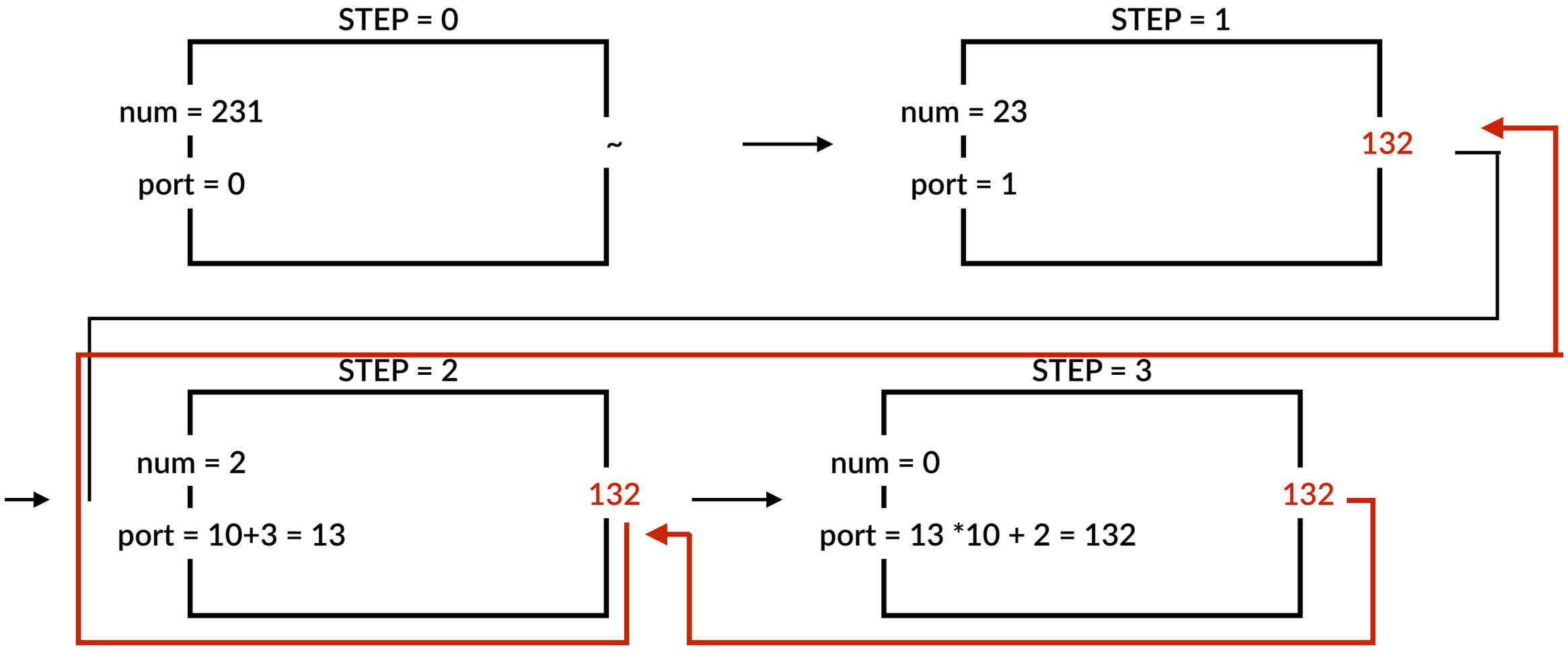

```
int funzione (int num, int port)
{
  if (num == 0)
    return port;
  else
    return funzione (num/10, port*10 + num%10);
}
```

Ricorsione - Esercizio III - *Soluzione*



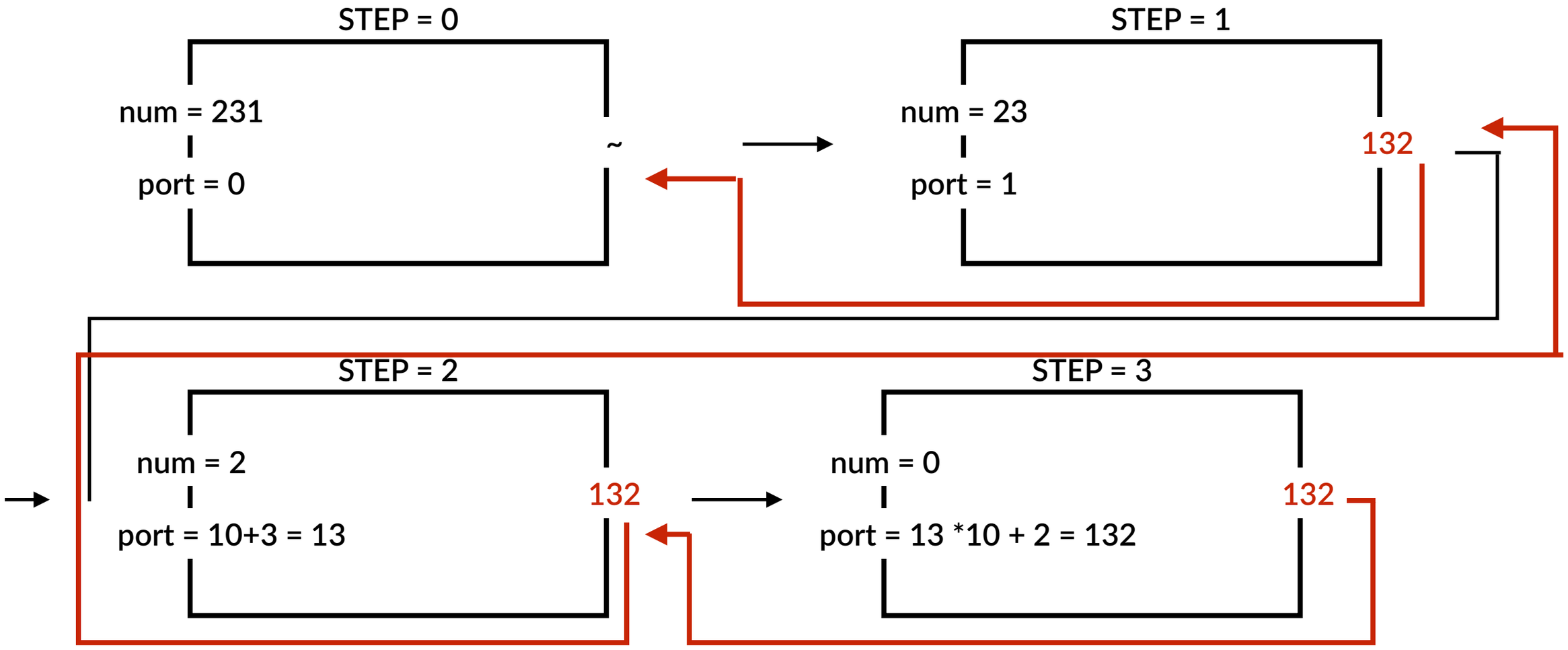

```
int funzione (int num, int port)
{
  if (num == 0)
    return port;
  else
    return funzione (num/10, port*10 + num%10);
}
```

Ricorsione - Esercizio III - *Soluzione*



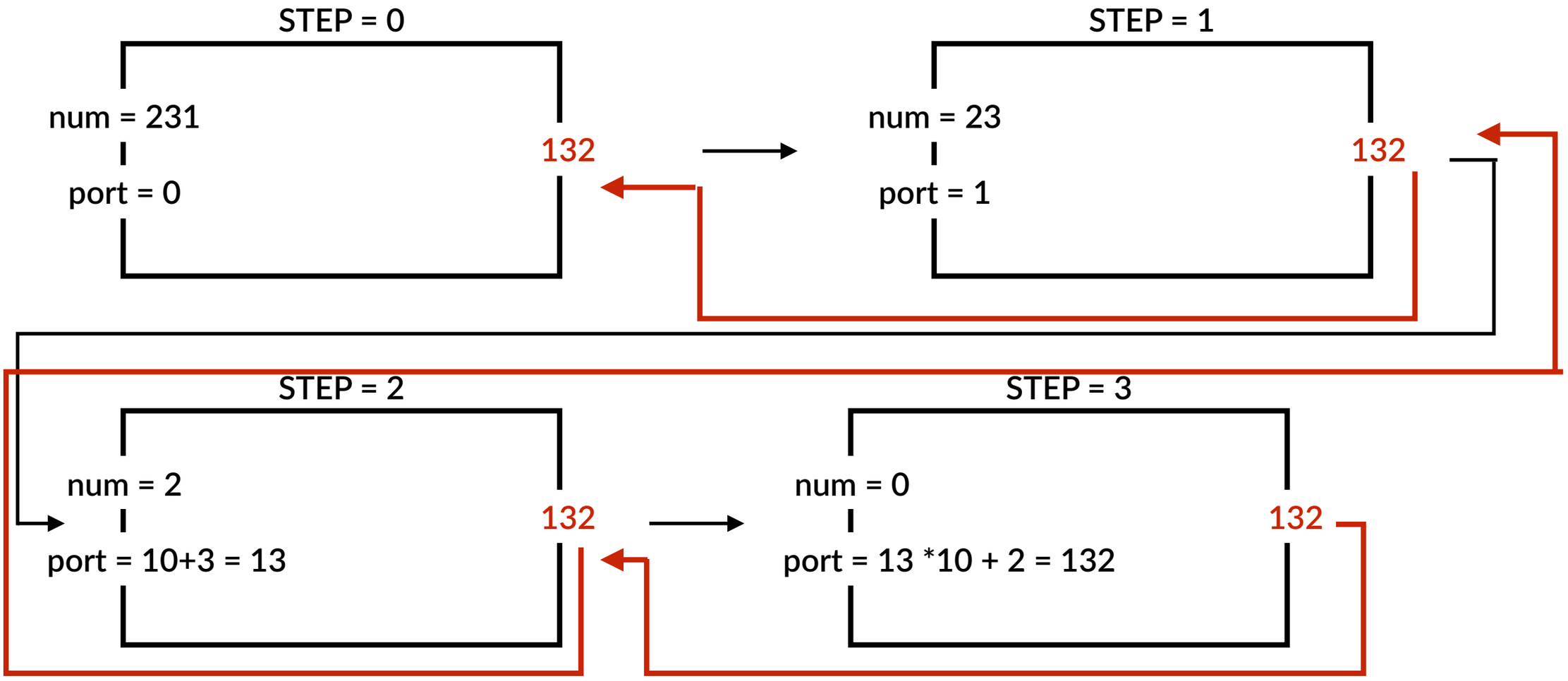

```
int funzione (int num, int port)
{
  if (num == 0)
    return port;
  else
    return funzione (num/10, port*10 + num%10);
}
```

Ricorsione - Esercizio III - *Soluzione*




```
int funzione (int num, int port)
{
  if (num == 0)
    return port;
  else
    return funzione (num/10, port*10 + num%10);
}
```

Ricorsione - Esercizio III - *Soluzione*




```
int funzione (int num, int port)
{
  if (num == 0)
    return port;
  else
    return funzione (num/10, port*10 + num%10);
}
```

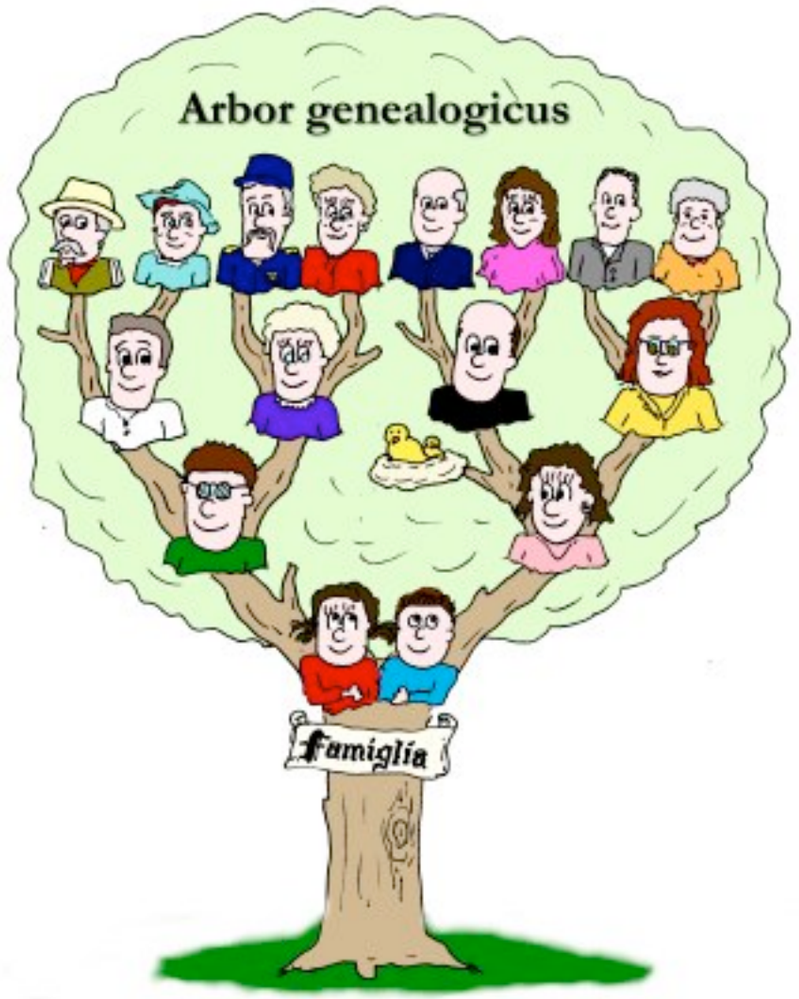
Un esercizio complesso:

Albero genealogico

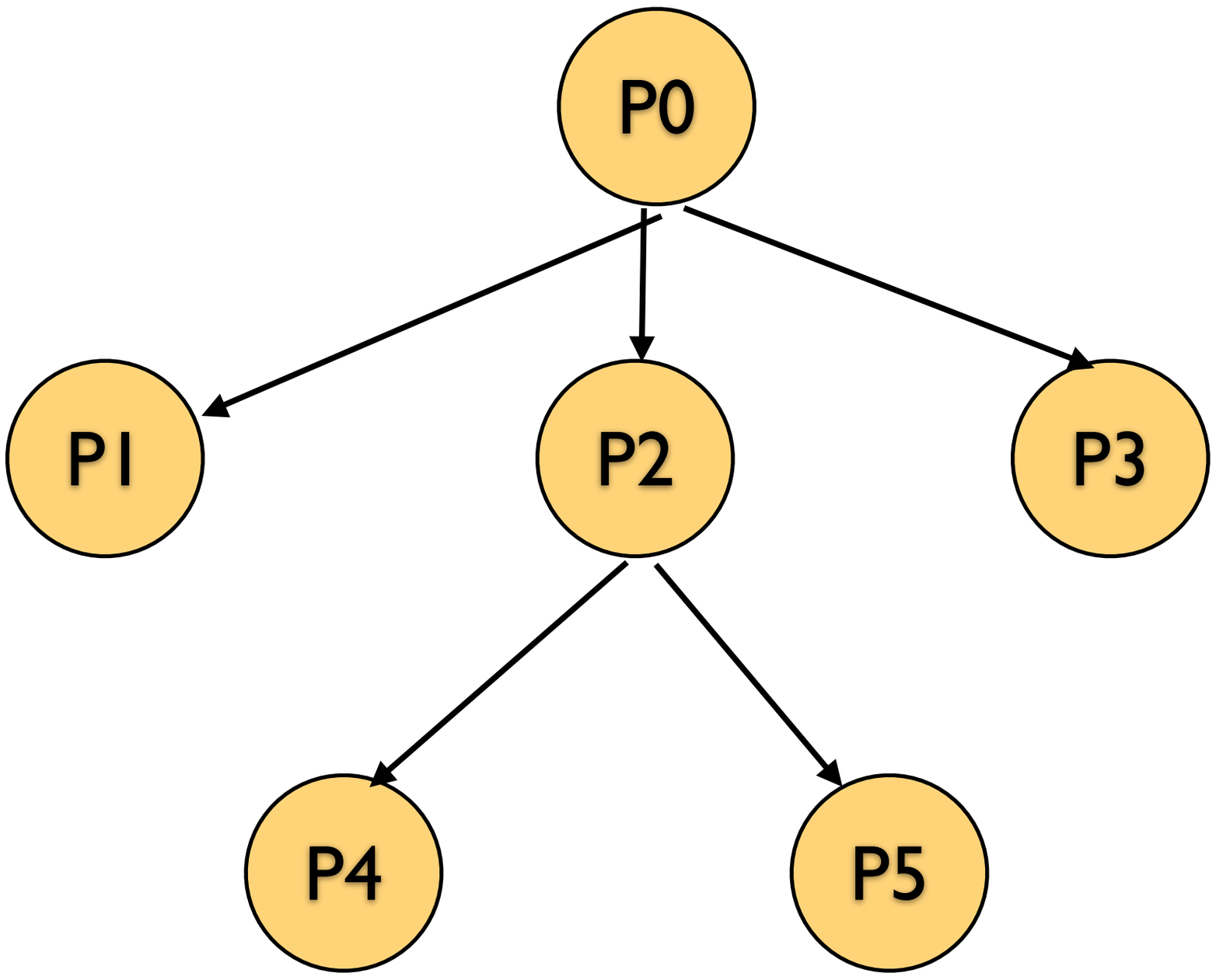
L'albero genealogico

- Scrivere un programma C che sia in grado di rappresentare e gestire un albero genealogico

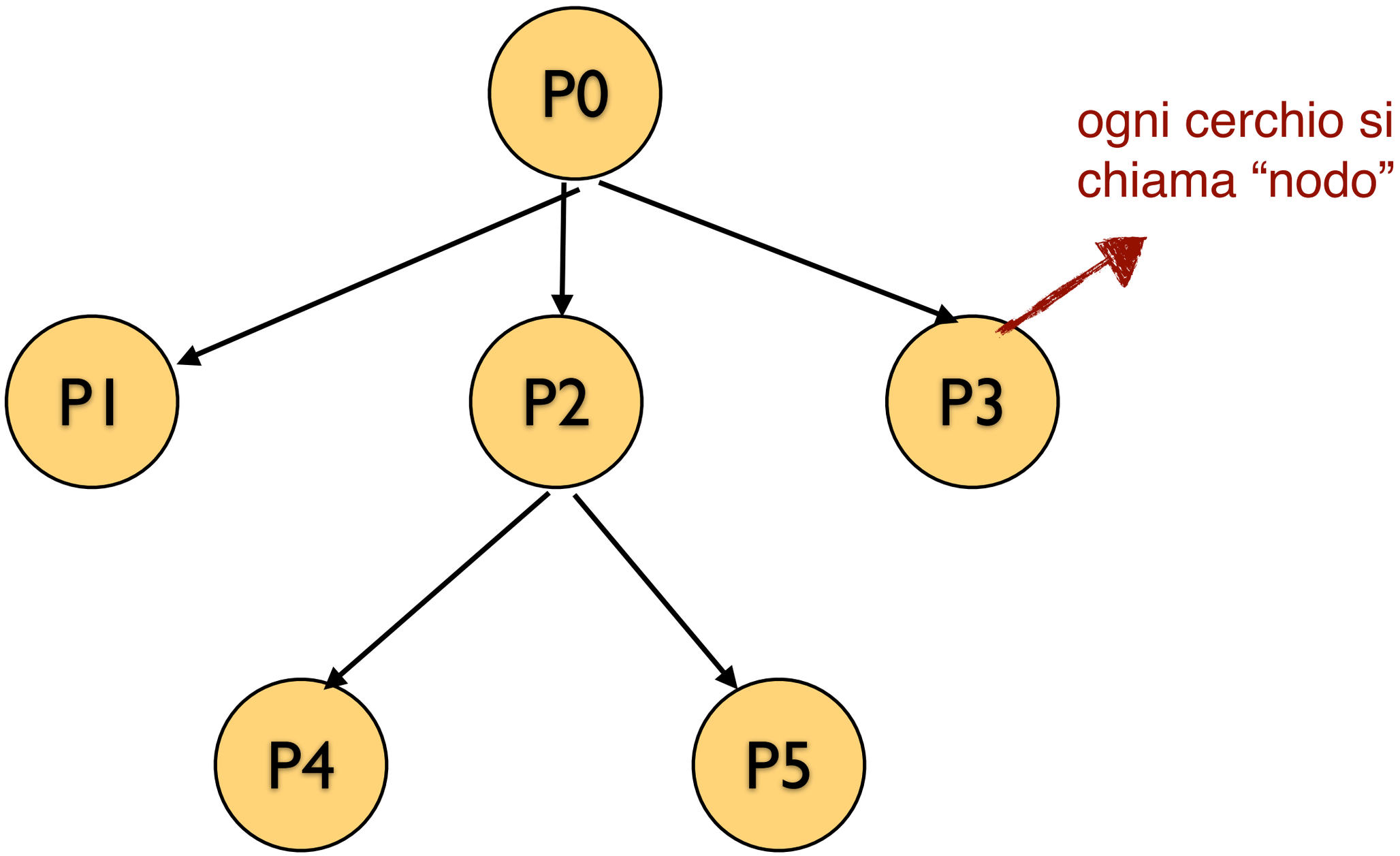
- In particolare, vogliamo poter fare:
  - Creare una persona
  - Rappresentare di una popolazione
  - Aggiungere figli ad una persona
  - Elencare i figli e i nipoti dato un antenato



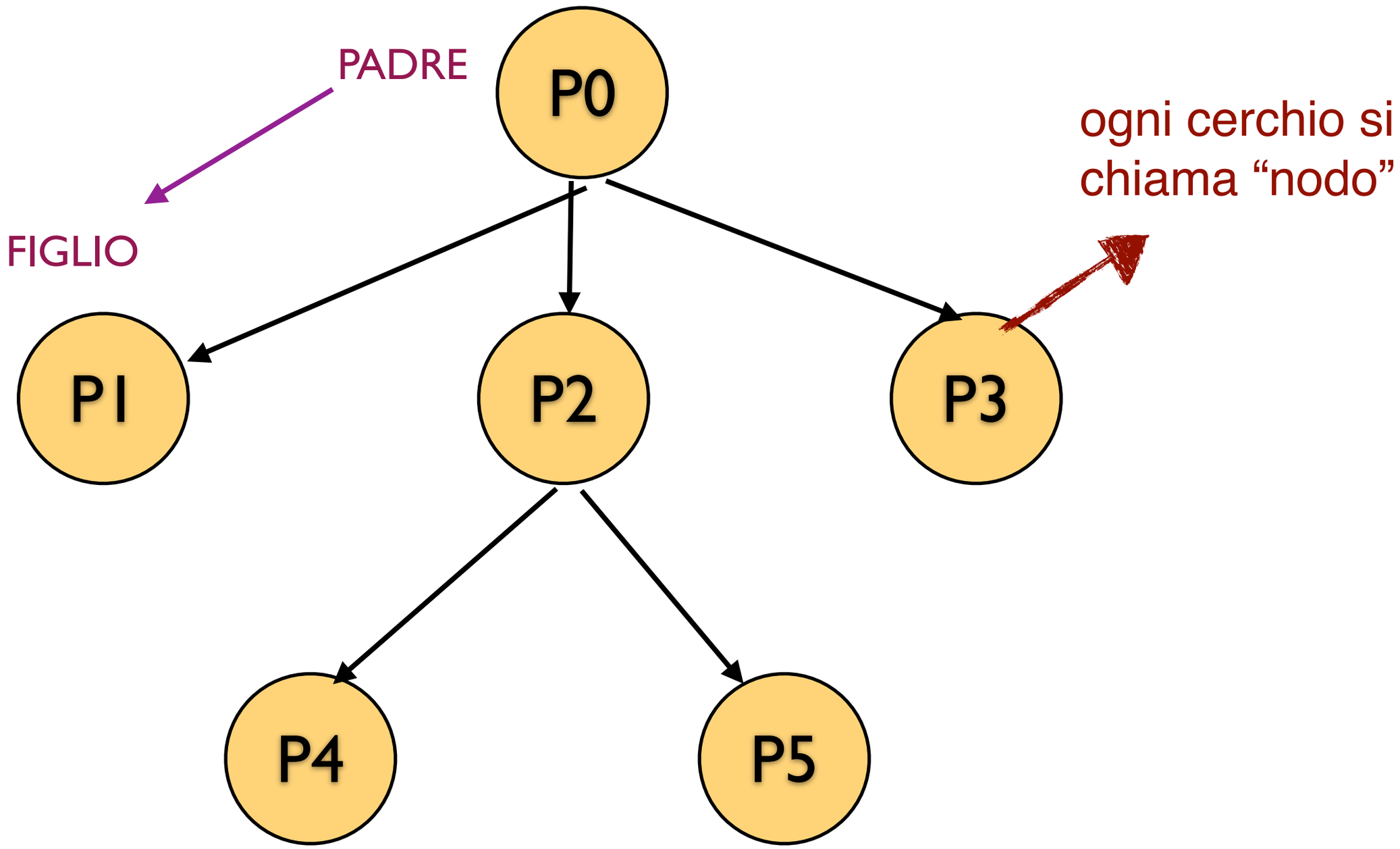
Una famosa struttura dati: l'albero



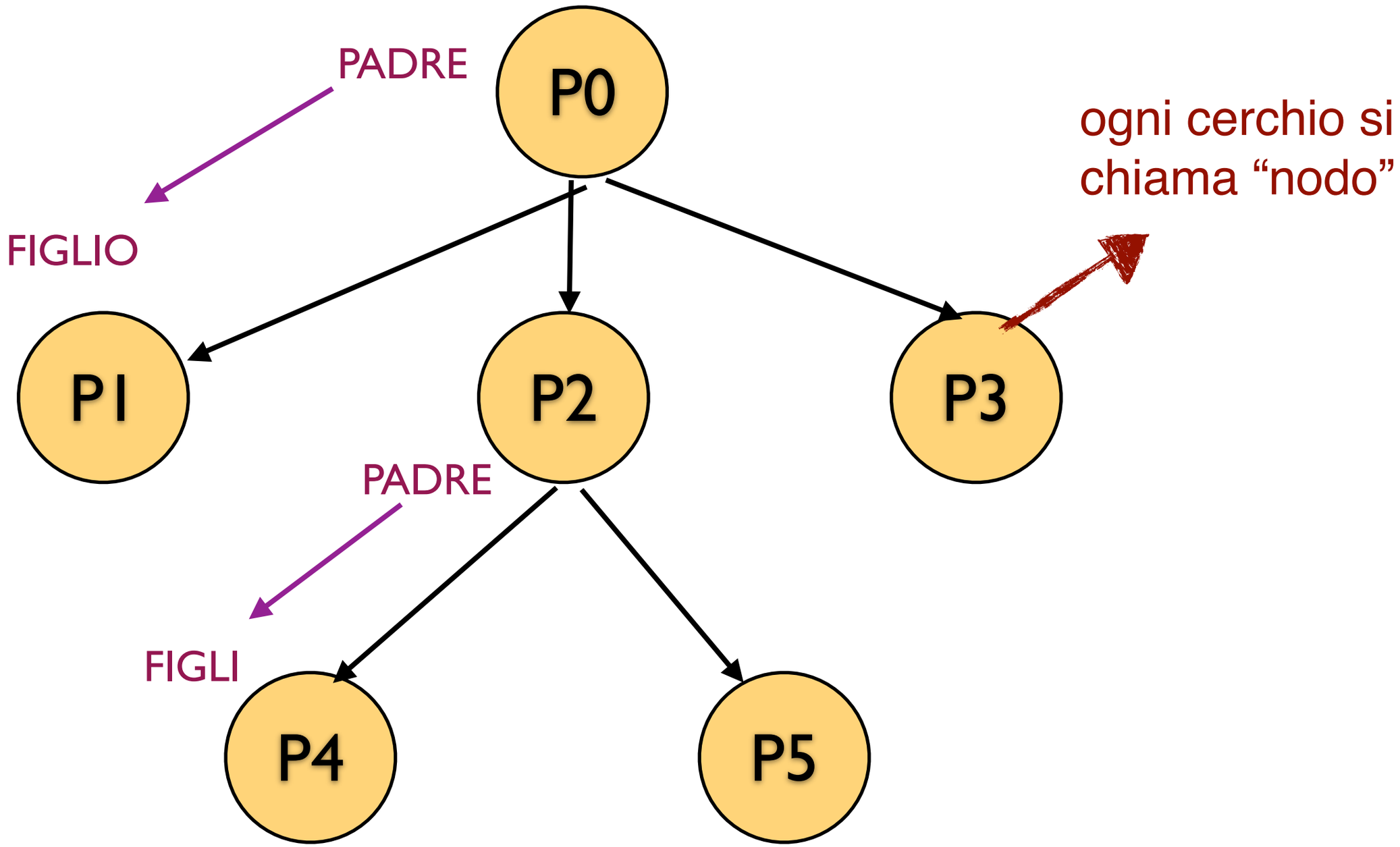
Una famosa struttura dati: l'albero



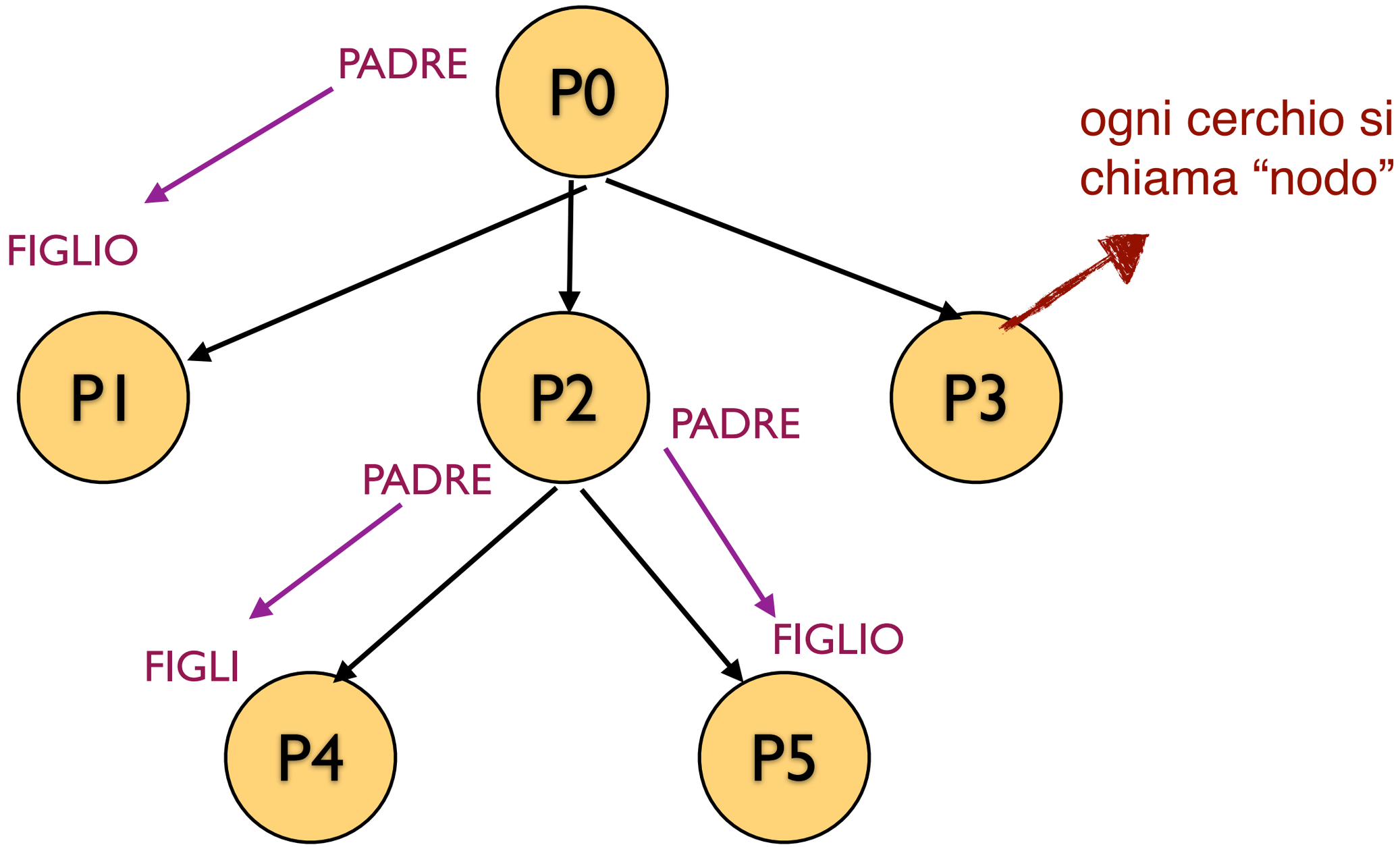
Una famosa struttura dati: l'albero



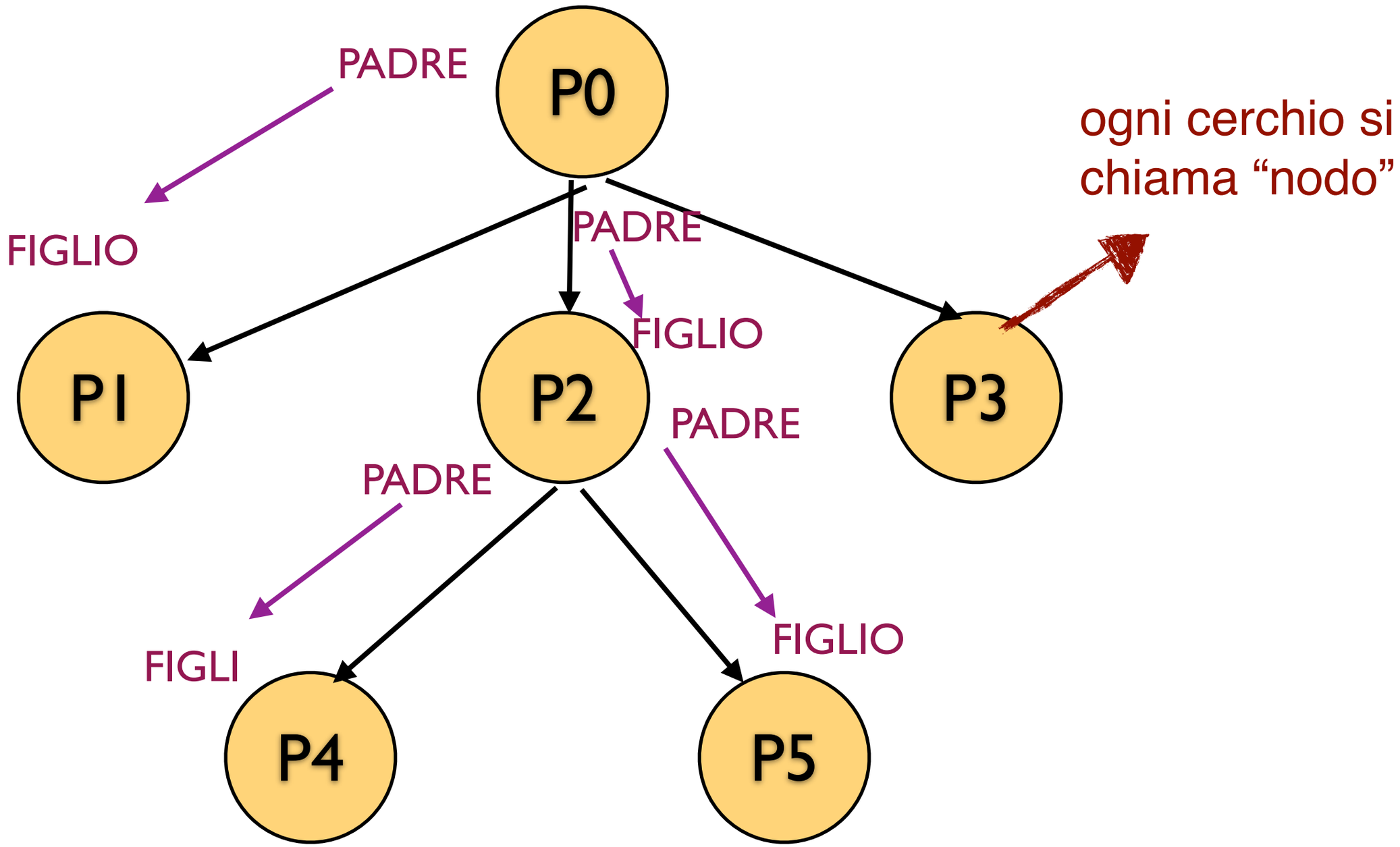
Una famosa struttura dati: l'albero



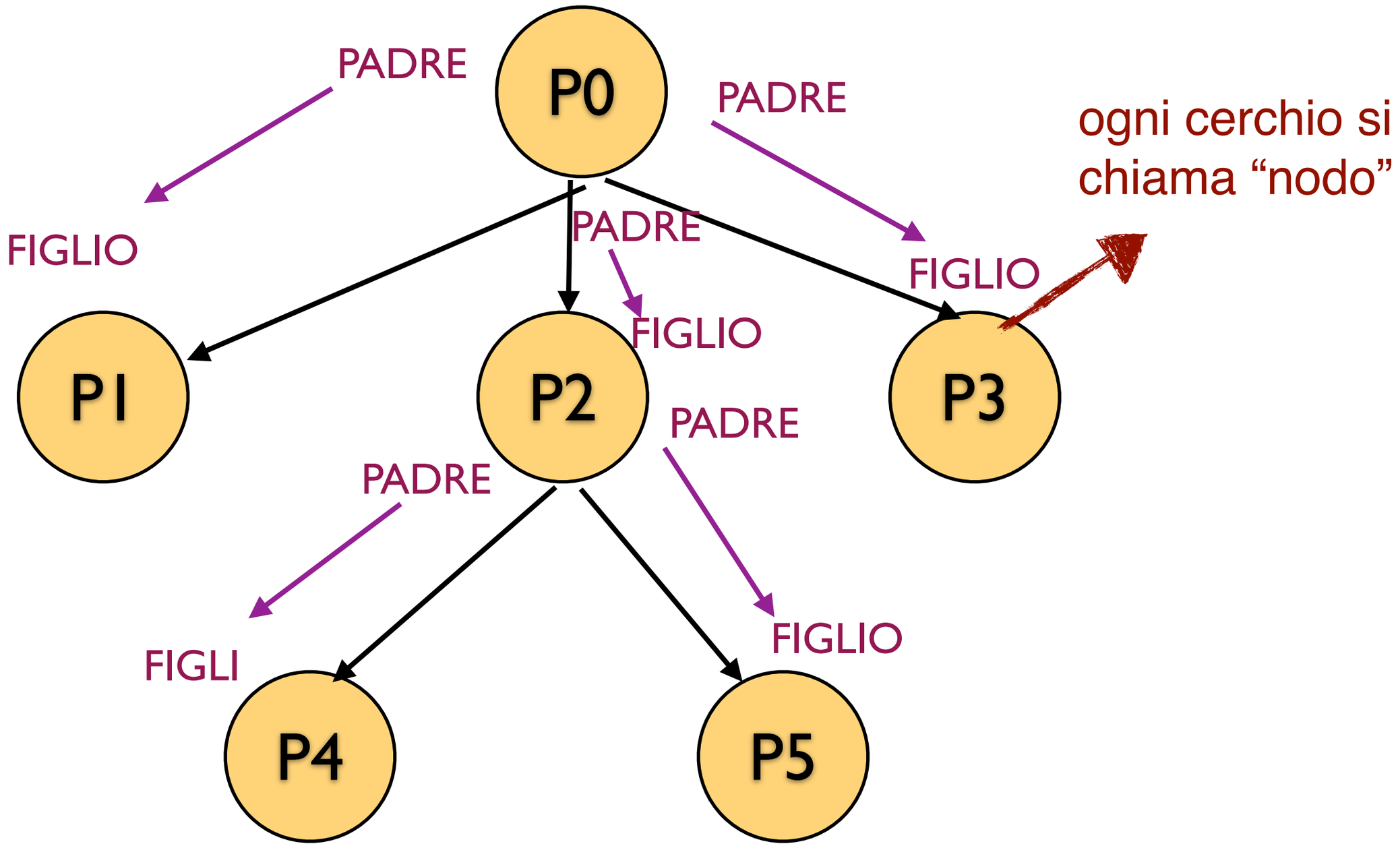
Una famosa struttura dati: l'albero



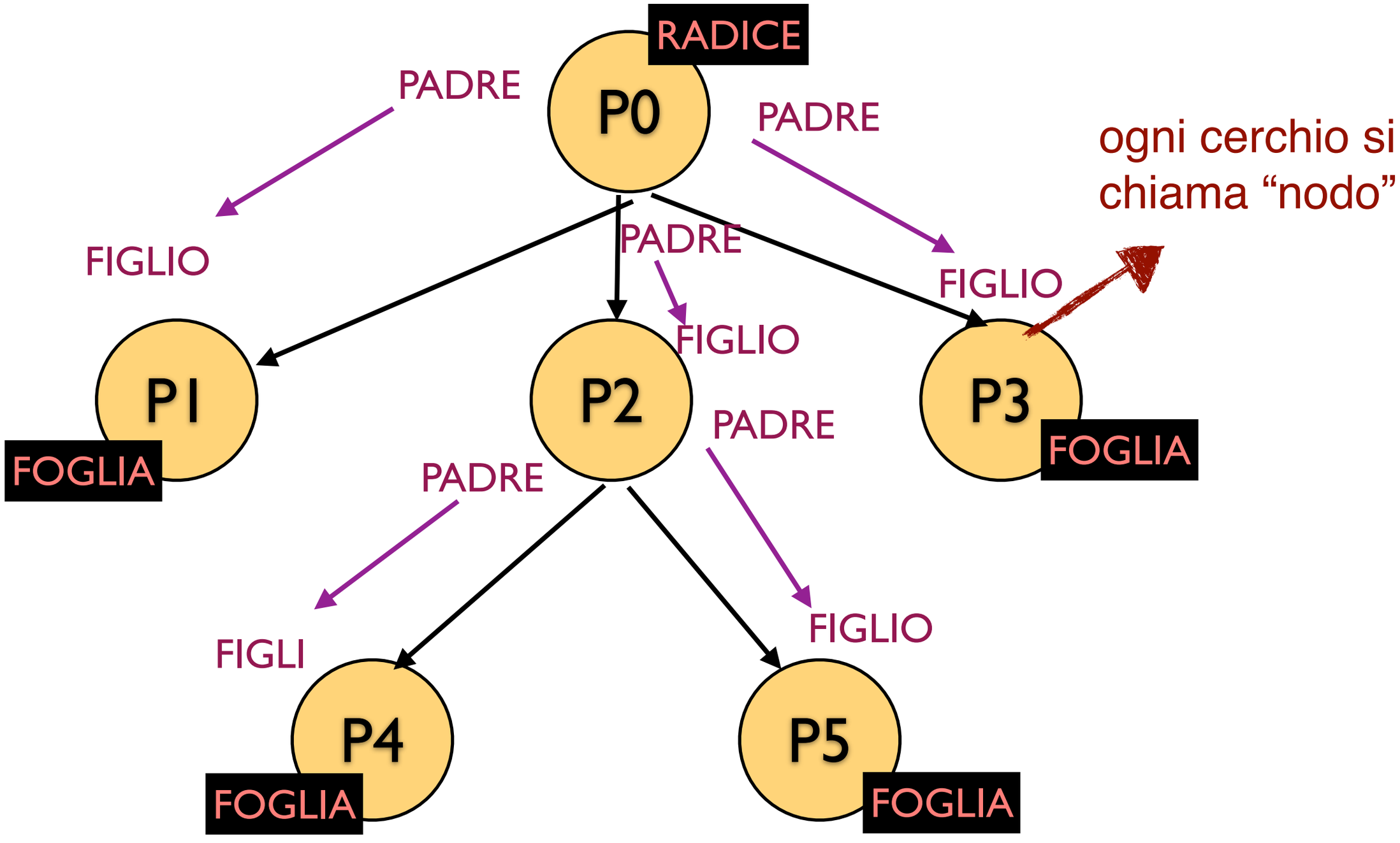
Una famosa struttura dati: l'albero



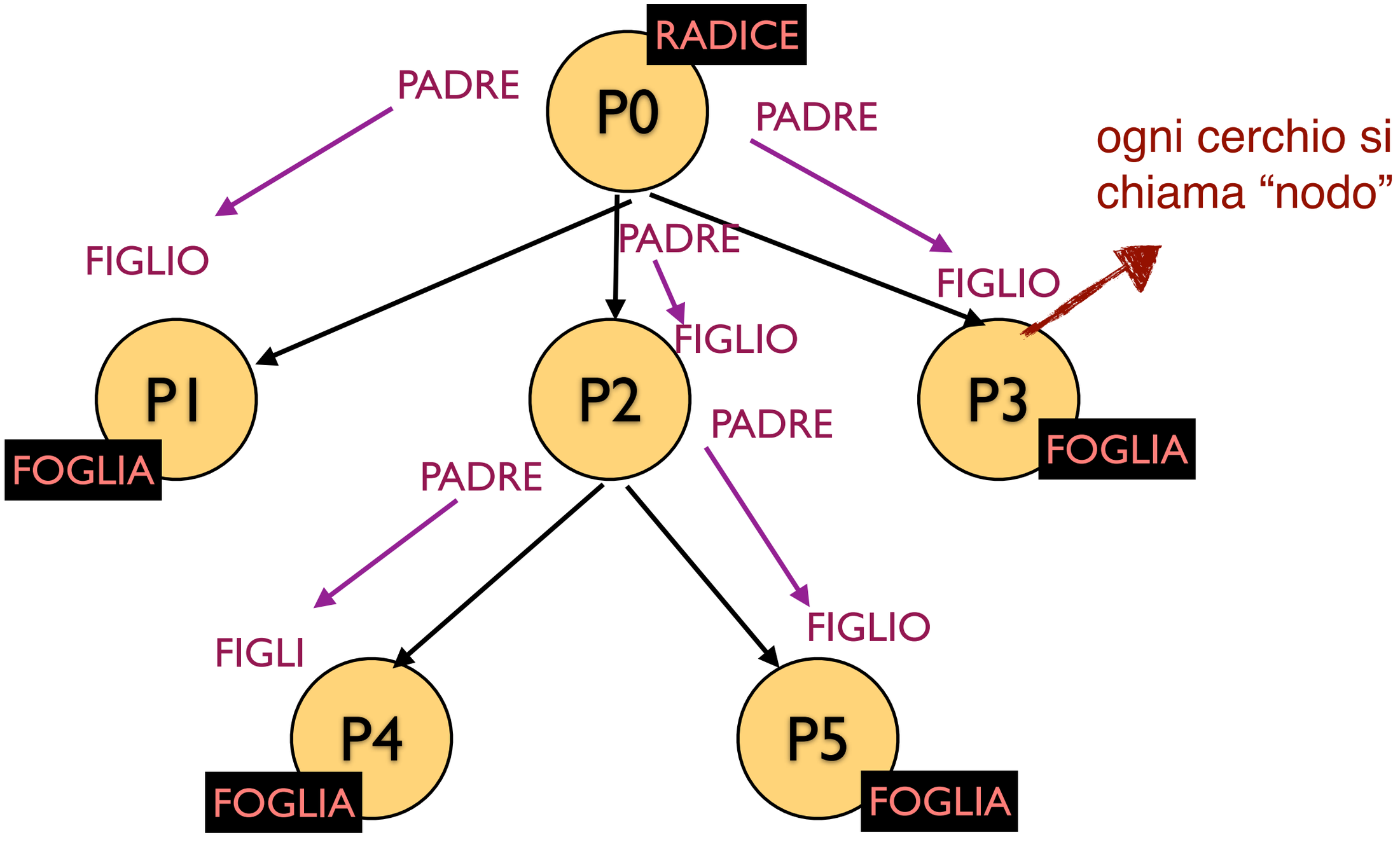
Una famosa struttura dati: l'albero



Una famosa struttura dati: l'albero

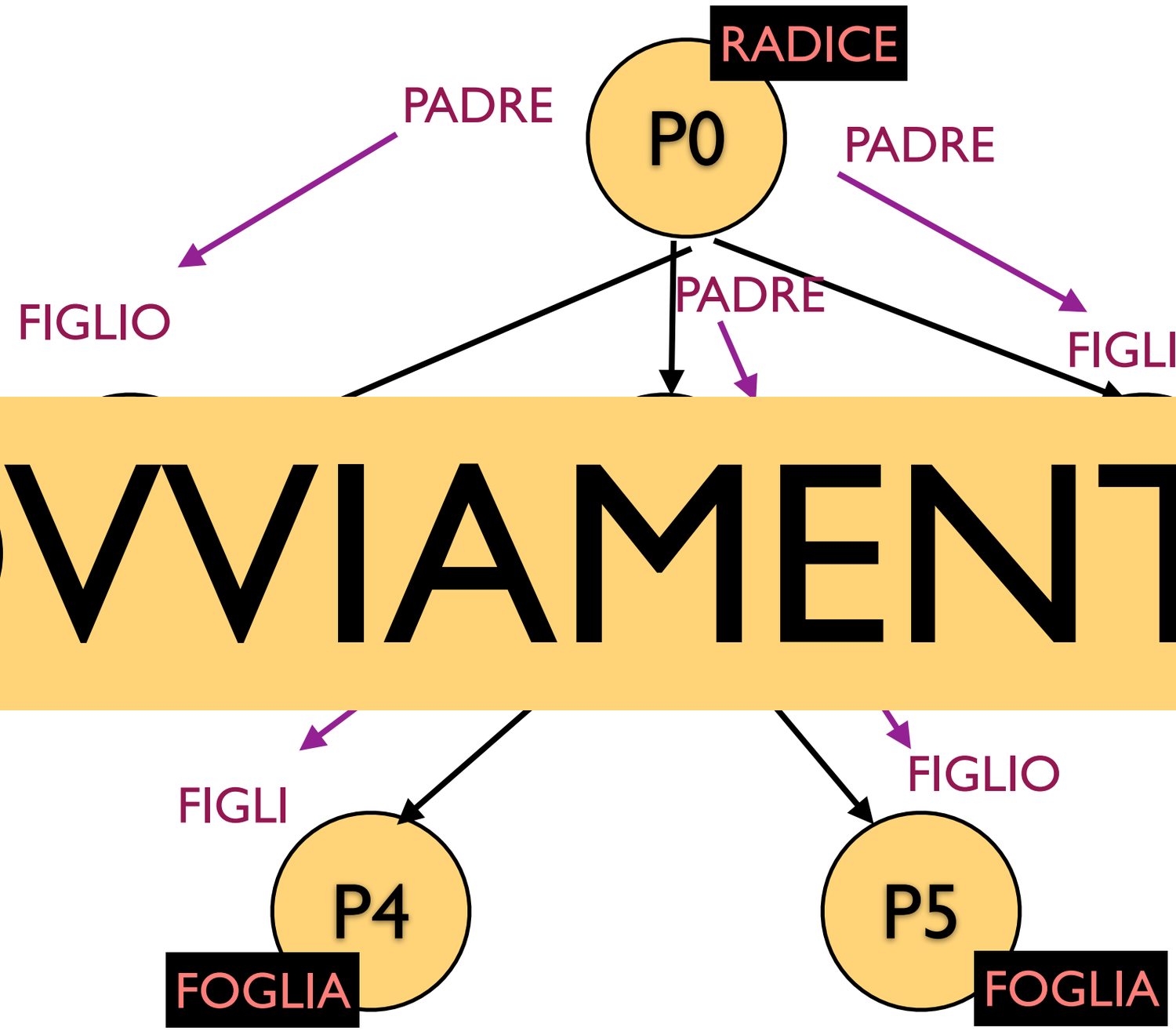


Una famosa struttura dati: l'albero



Può essere utile per rappresentare un albero genealogico?

Una famosa struttura dati: l'albero



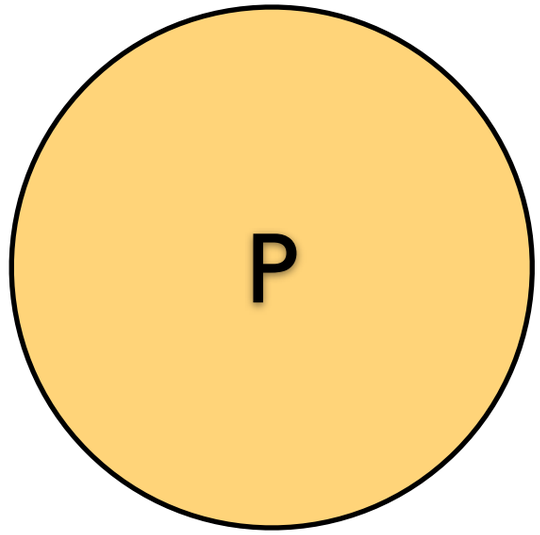
ogni cerchio si chiama "nodo"

OVVIAMENTE SI

Può essere utile per rappresentare un albero genealogico?

- OGNI NODO DELL'ALBERO SARA' PER NOI UNA PERSONA

==

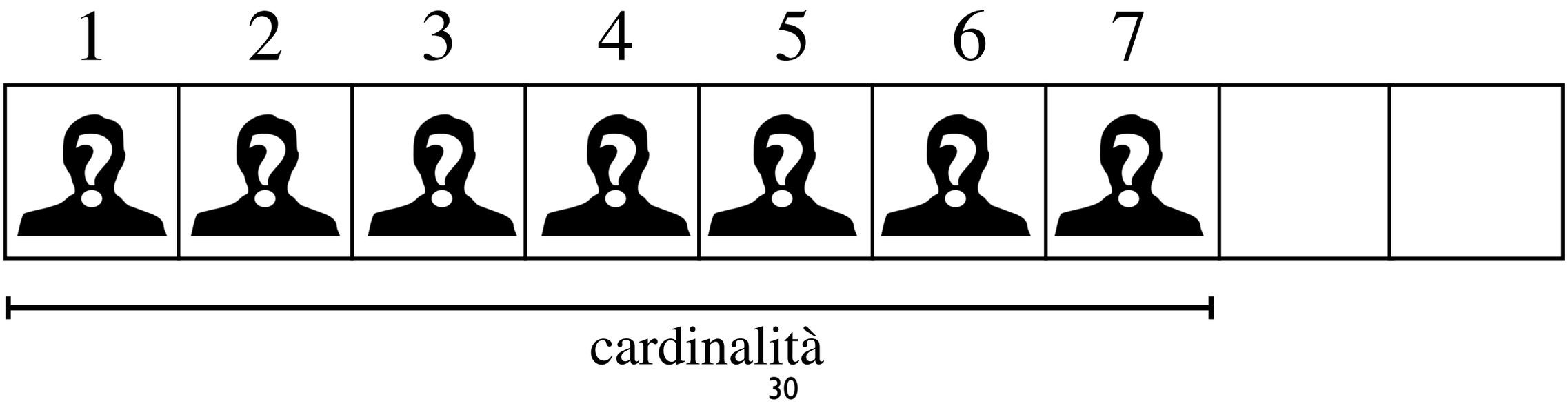


Una Persona

- SESSO
- NOME
- ETA?
- CHI SONO I GENITORI?
- CHI SONO I FIGLI?
- QUANTI FIGLI?

Una Popolazione

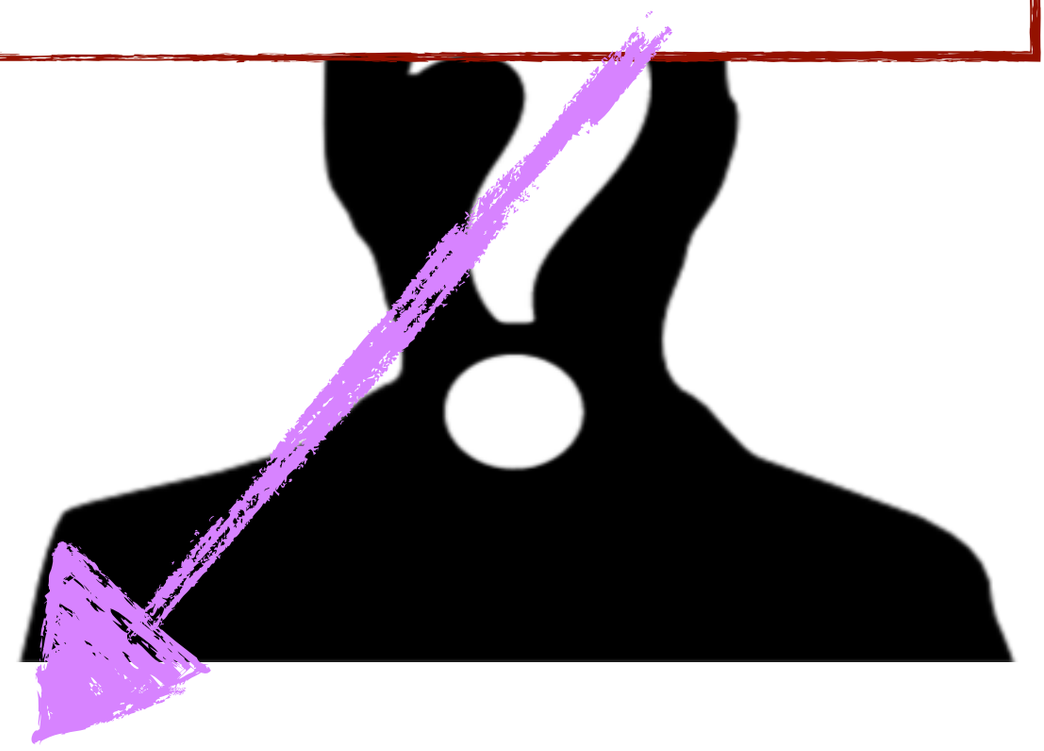
- Una popolazione è rappresentata da un insieme di persone
- Ogni persona ha un suo indice (numero univoco di identificazione)
- Esiste un numero di persone della

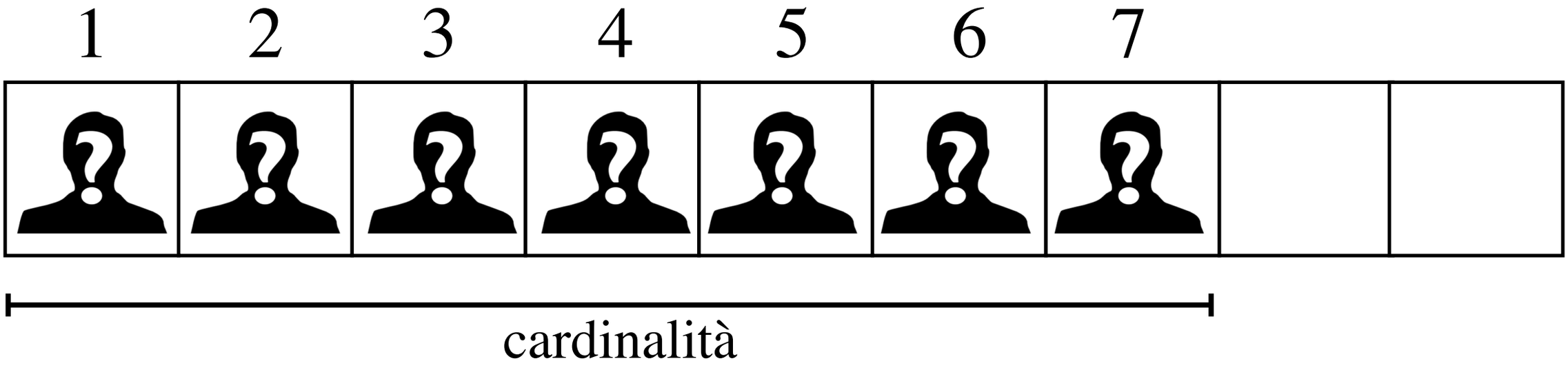


Una Persona nella popolazione

- SESSO
- NOME
- ETA?
- CHI SONO I GENITORI?
- CHI SONO I FIGLI?
- QUANTI FIGLI?

Li rappresentiamo con l'indice della persona nella popolazione





Persona e Popolazione (codice C)

```c
typedef struct {

} persona;
```

```c
int main () {

    persona* popolazione[MAX_PERSONE];
    int cardinalita_popolazione = -1;
```

Persona e Popolazione (codice C)

```c
typedef struct {
    genere sesso;
    char nome[STR_LEN];
    int i_persona;
    int i_genitore1;
    int i_genitore2;
    int i_figli[MAX_FIGLI];
    int numero_figli;
    int eta;
} persona;
```

```c
int main () |
{

    persona* popolazione[MAX_PERSONE];
    int cardinalita_popolazione = -1;
```

Persona e Popolazione (codice C)

```c
typedef enum {MASCHIO, FEMMINA} genere;

typedef struct {
    genere sesso;
    char nome[STR_LEN];
    int i_persona;
    int i_genitore1;
    int i_genitore2;
    int i_figli[MAX_FIGLI];
    int numero_figli;
    int eta;
} persona;
```

```c
int main () |
{

    persona* popolazione[MAX_PERSONE];
    int cardinalita_popolazione = -1;
```

Creazione di una persona

```
persona crea_persona(genere sesso, char nome[STR_LEN], int eta)
{

}
```

Creazione di una persona

```
persona crea_persona(genere sesso, char nome[STR_LEN], int eta)
{
    persona p;
    p.sesso = sesso;
    strcpy(p.nome,nome);
    p.i_genitore1 = -1;
    p.i_genitore2 = -1;
    p.numero_figli = 0;
    p.eta = eta;
    return p;
}
```

Aggiunta persona alla popolazione

```
int aggiungi_a_popolazione(persona* popolazione[MAX_PERSONE],
    persona* p, int* cardinalita_popolazione)
{

}
```

Aggiunta persona alla popolazione

```
int aggiungi_a_popolazione(persona* popolazione[MAX_PERSONE],
    persona* p, int* cardinalita_popolazione)
{
    (*cardinalita_popolazione)++;
    p->i_persona = *cardinalita_popolazione;
    popolazione[*cardinalita_popolazione] = p;
}
```

Aggiunta di un figlio

```
void aggiungi_figlio(persona* genitore, persona* figlio)
{

}
```

Aggiunta di un figlio

```
void aggiungi_figlio(persona* genitore, persona* figlio)
{
    genitore->i_figli[genitore->numero_figli] = figlio->i_persona;
    genitore->numero_figli++;

    if (figlio->i_genitore1 == -1)
        figlio->i_genitore1 = genitore->i_persona;
    else if (figlio->i_genitore2 == -1)
        figlio->i_genitore2 = genitore->i_persona;
    else
        printf("errore\n");

}
```

Funzioni di stampa a schermo

```
char* genere_to_str(genere sesso)
{



}

void stampa_persona(persona* p)
{


}
```

Funzioni di stampa a schermo

```
char* genere_to_str(genere sesso)
{
    if (sesso == MASCHIO) return "MASCHIO";
    if (sesso == FEMMINA) return "FEMMINA";
    return "?";
}

void stampa_persona(persona* p)
{


}
```

Funzioni di stampa a schermo

```
char* genere_to_str(genere sesso)
{
    if (sesso == MASCHIO) return "MASCHIO";
    if (sesso == FEMMINA) return "FEMMINA";
    return "?";
}

void stampa_persona(persona* p)
{
    printf("Nome: %s - Genere: %s - Eta':%d - Id:%d - #Figli: %d\n",
    p->nome, genere_to_str(p->sesso), p->eta, p->i_persona, p->numero_figli);
}
```


That's all Folks!

Tutte il materiale sarà disponibile sul mio sito internet!

alessandronacci.it