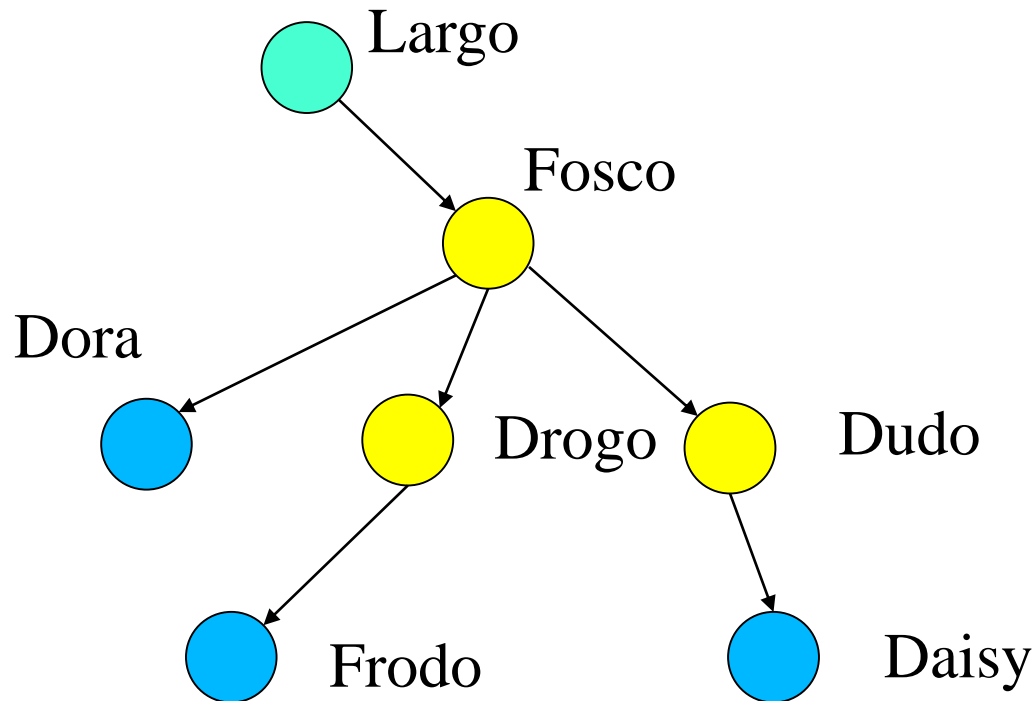


Alberi

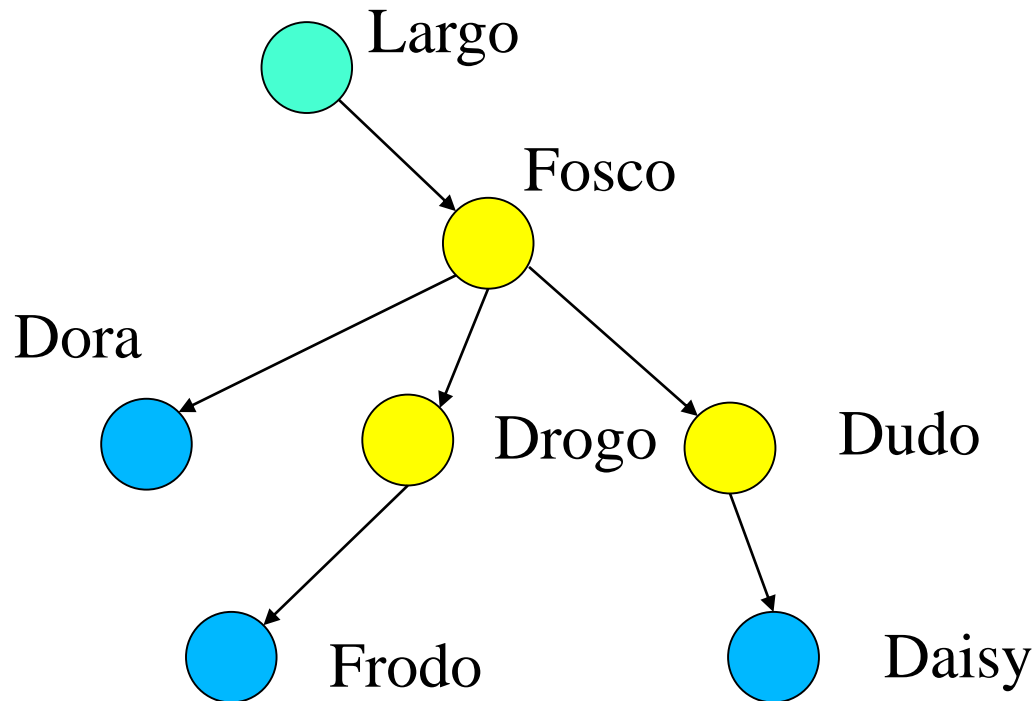
# Alberi

- Gli alberi sono una **generalizzazione** delle liste che consente di modellare delle strutture gerarchiche come questa:



# Alberi

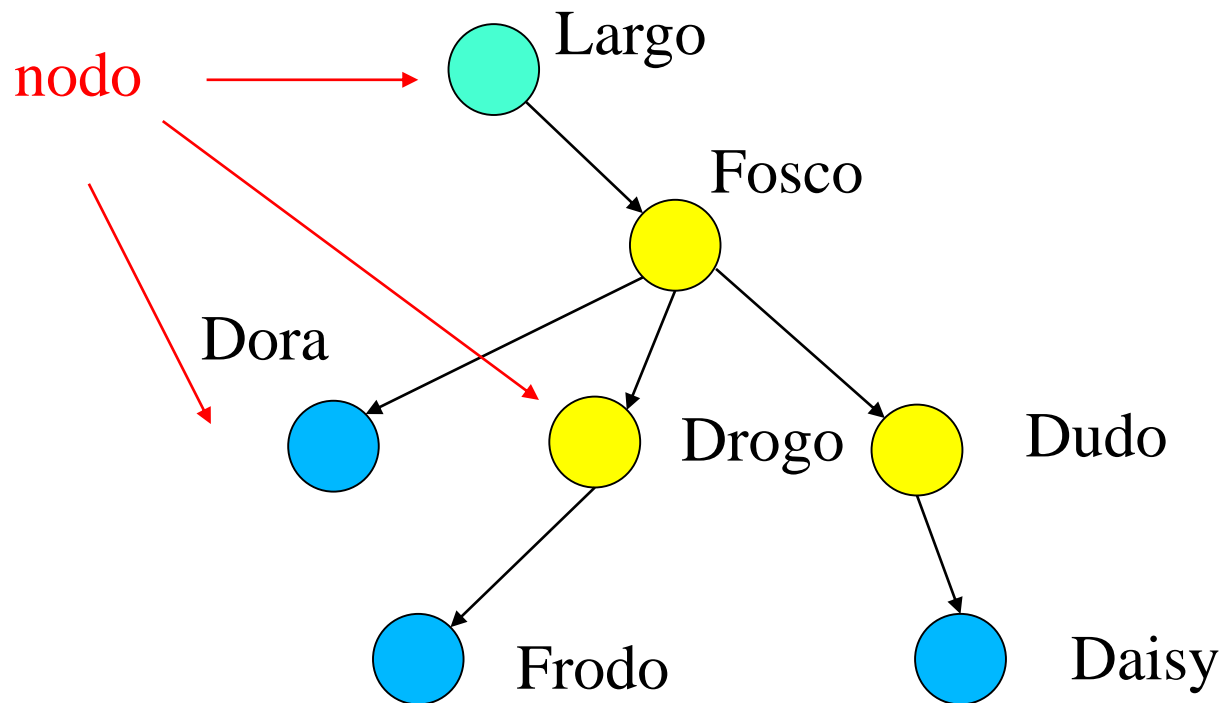
- Gli alberi sono una **generalizzazione** delle liste che consente di modellare delle strutture gerarchiche come questa:



L'albero genealogico della famiglia Baggins

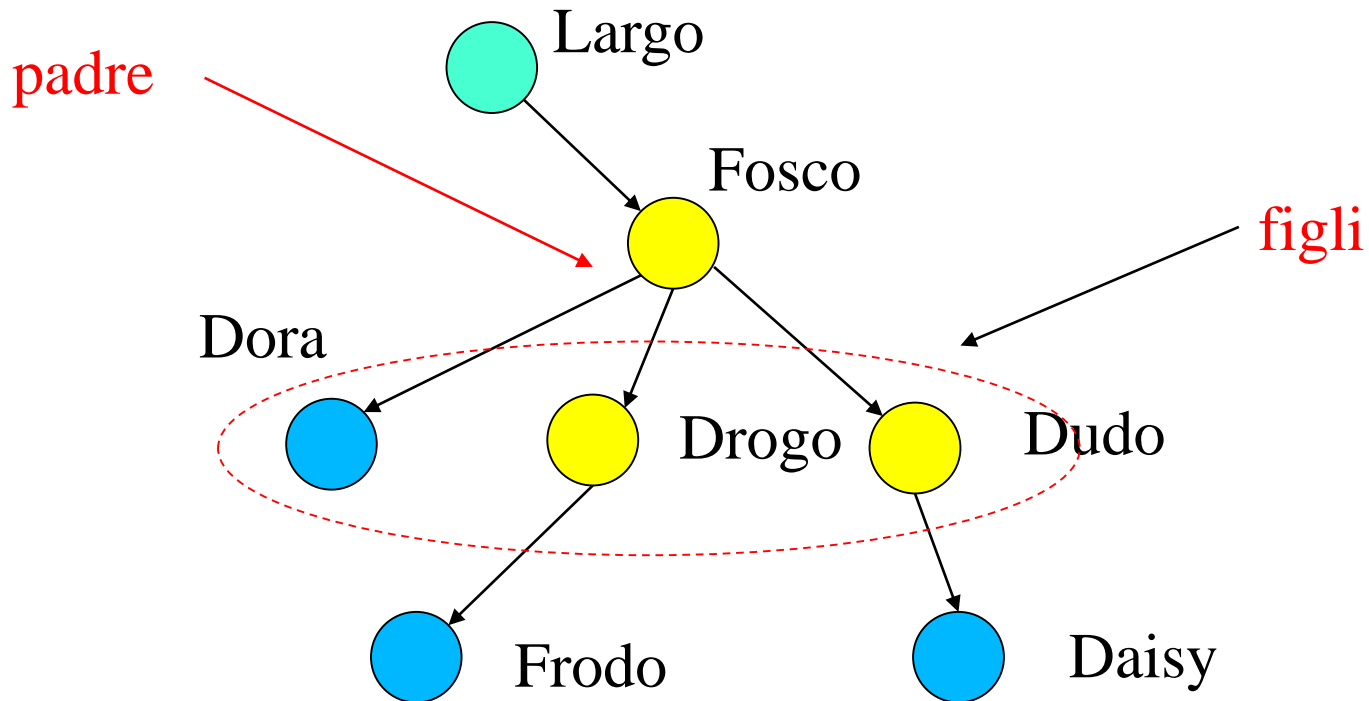
# Alberi

- Terminologia: gli elementi sono detti **nodi**



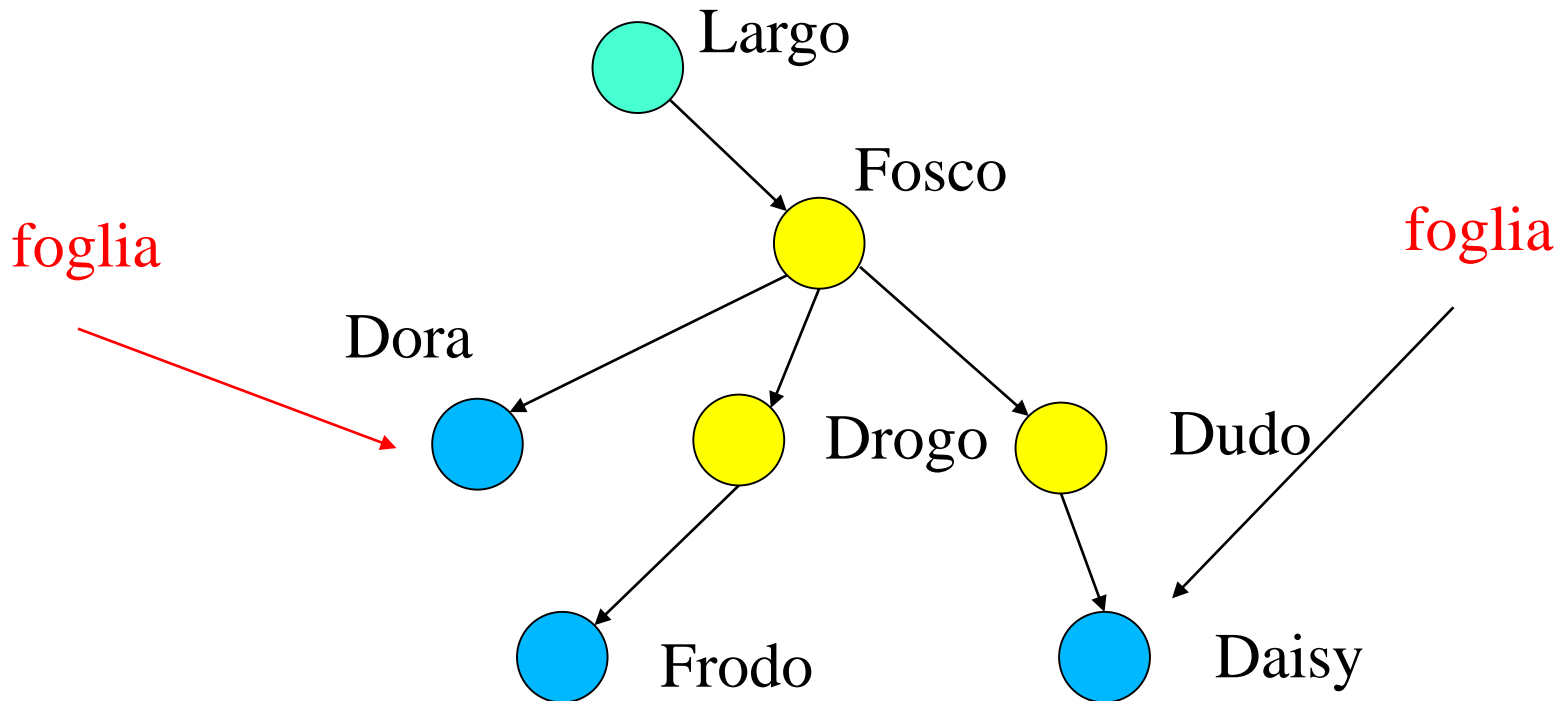
# Alberi

- i successori sono detti **figli** ed i predecessori **padre**



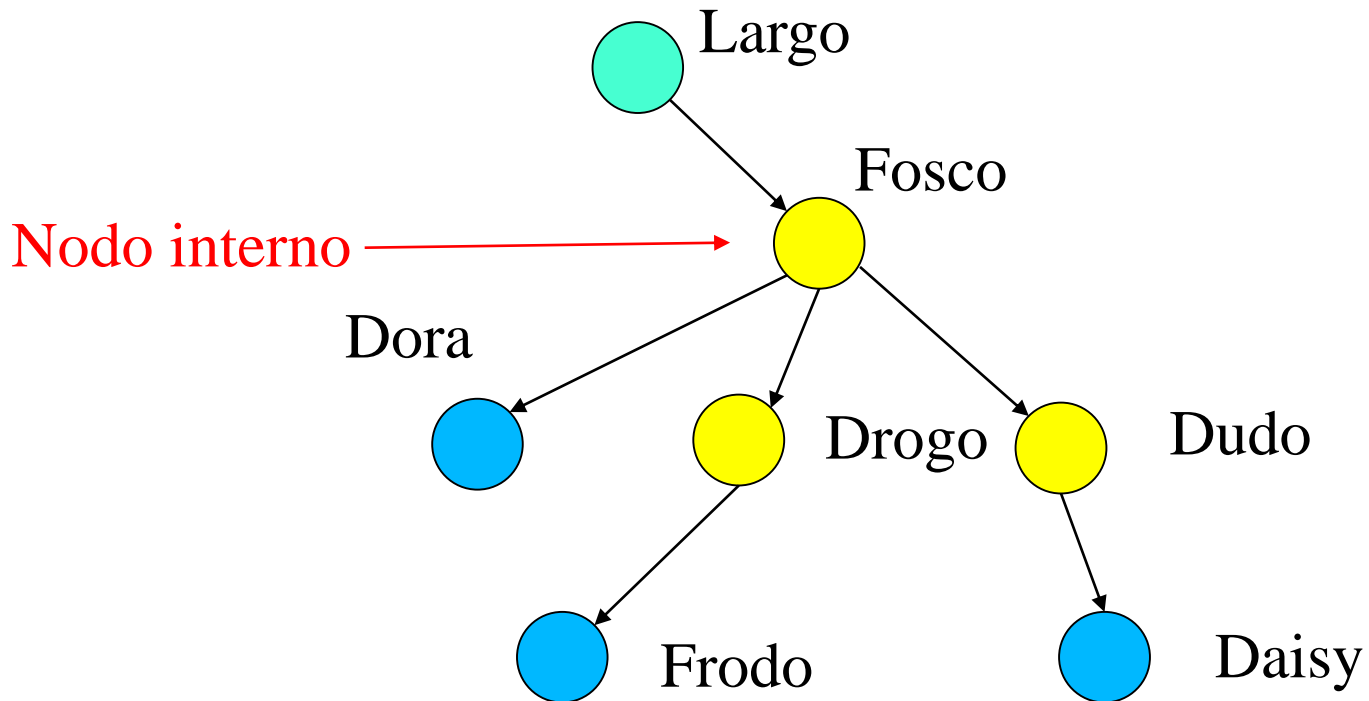
# Alberi

- i nodi senza figli sono detti **foglie** (in azzurro)



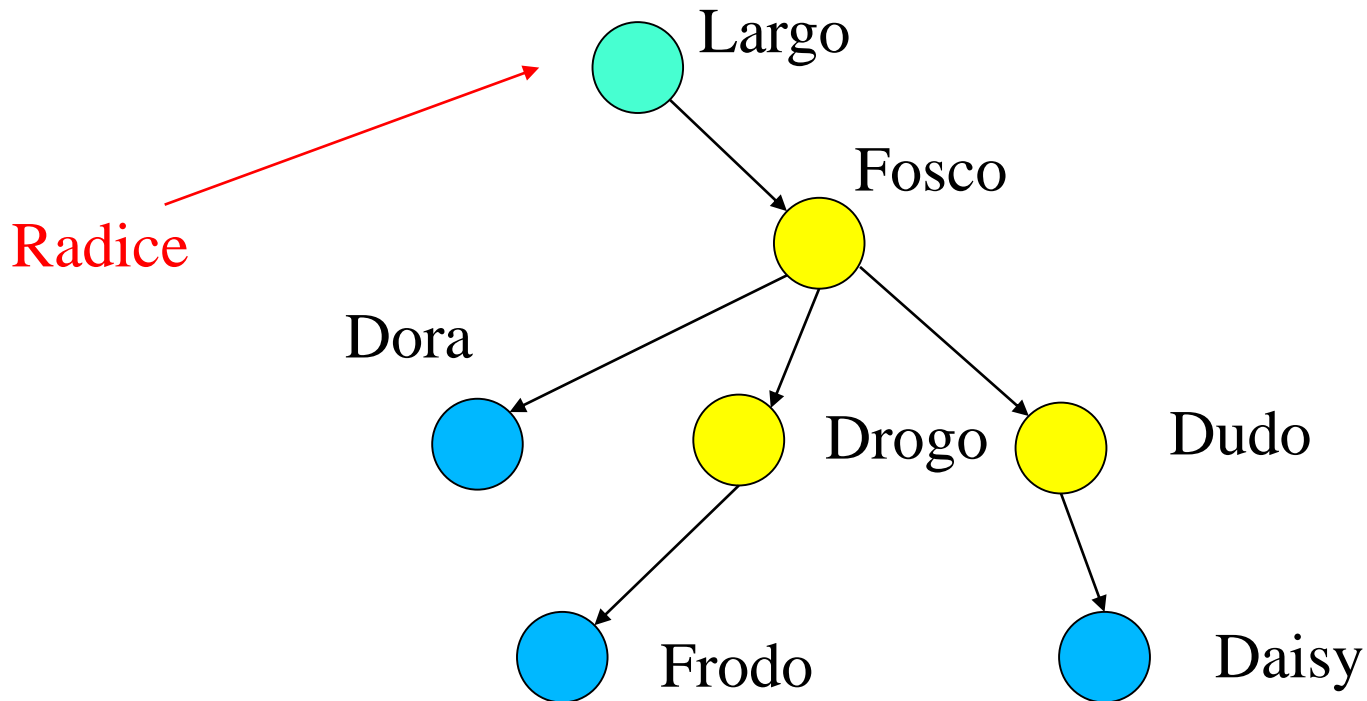
# Alberi

- i nodi con figli sono detti **interni** (in giallo)



# Alberi

- Il nodo iniziale è detto **radice** (in verde)



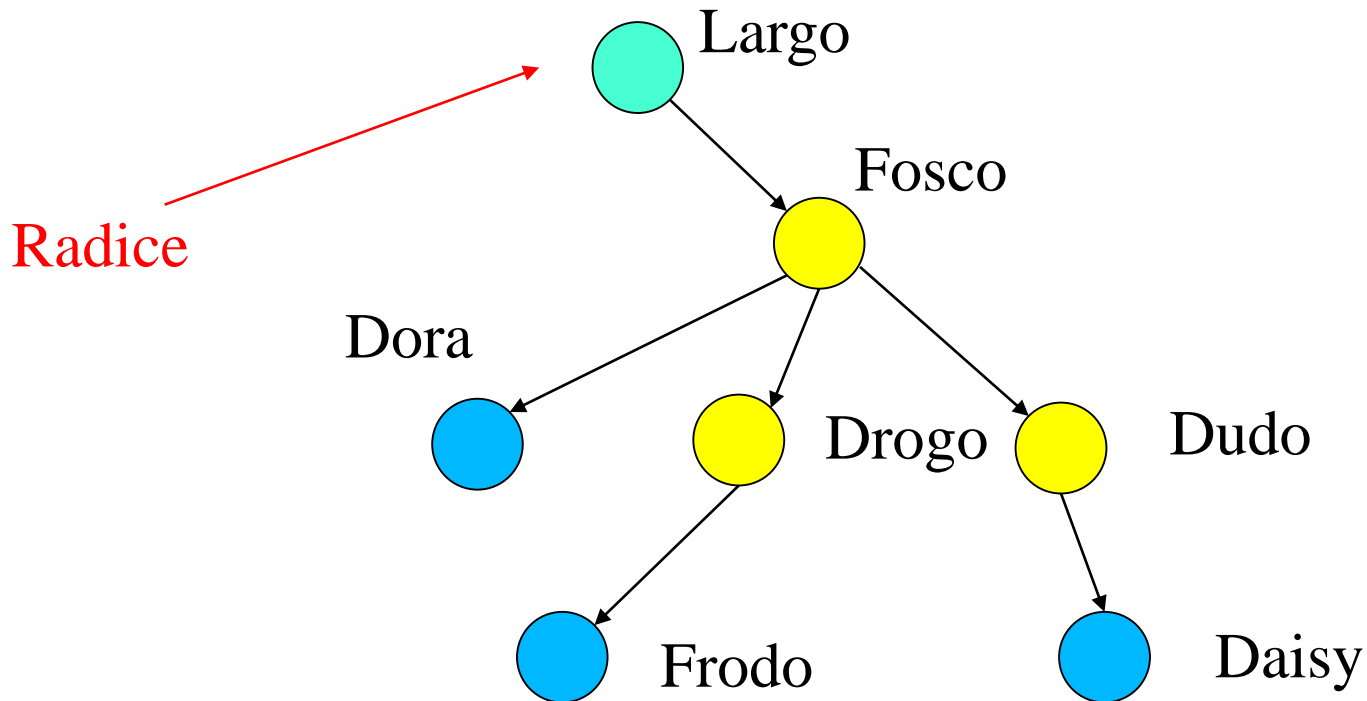


# Alcune definizioni

- **Cammino**
  - una sequenza di nodi, in cui ogni nodo è figlio del nodo che lo precede nella sequenza
- **Livello di un nodo**
  - numero di nodi da attraversare per arrivare dal nodo alla radice
  - Definizione induttiva:
    - La radice ha livello 0
    - Se un nodo ha livello  $n$  i suoi figli hanno livello  $n + 1$
- **Livello  $k$  dell'albero**
  - È l'insieme di tutti e soli i nodi di livello uguale a  $k$
- **Altezza dell'albero**
  - È il livello massimo dei nodi nell'albero

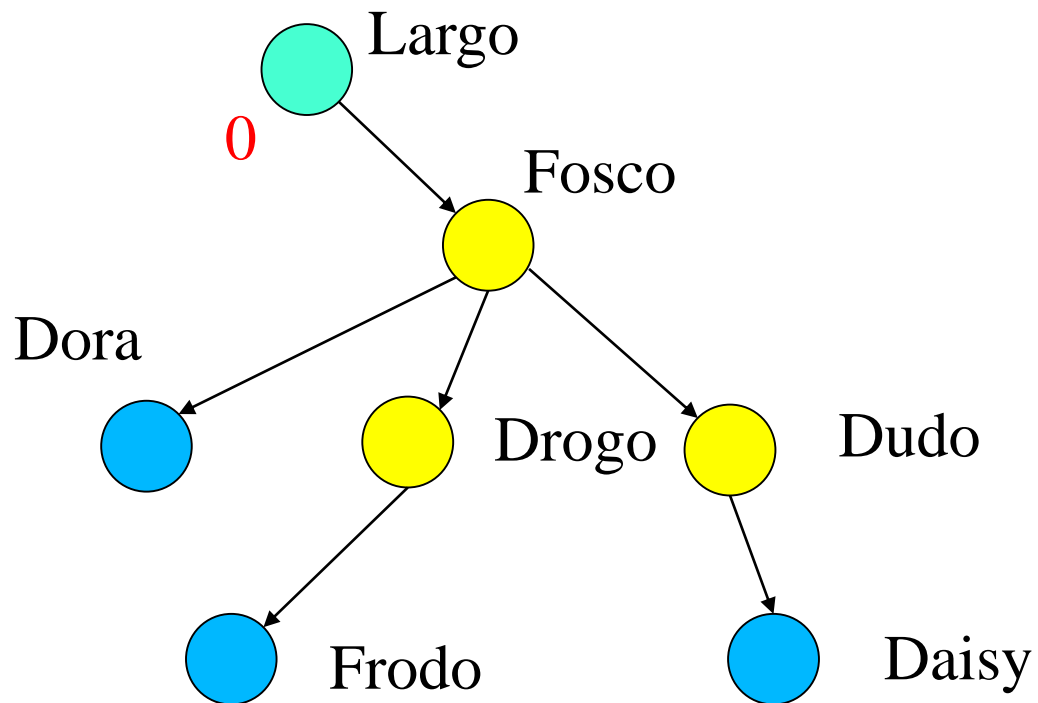
# Esempio

- Cammini
  - (Largo,Fosco,Dudo, Daisy) (Fosco,Dora) (Fosco,Drogo) ...



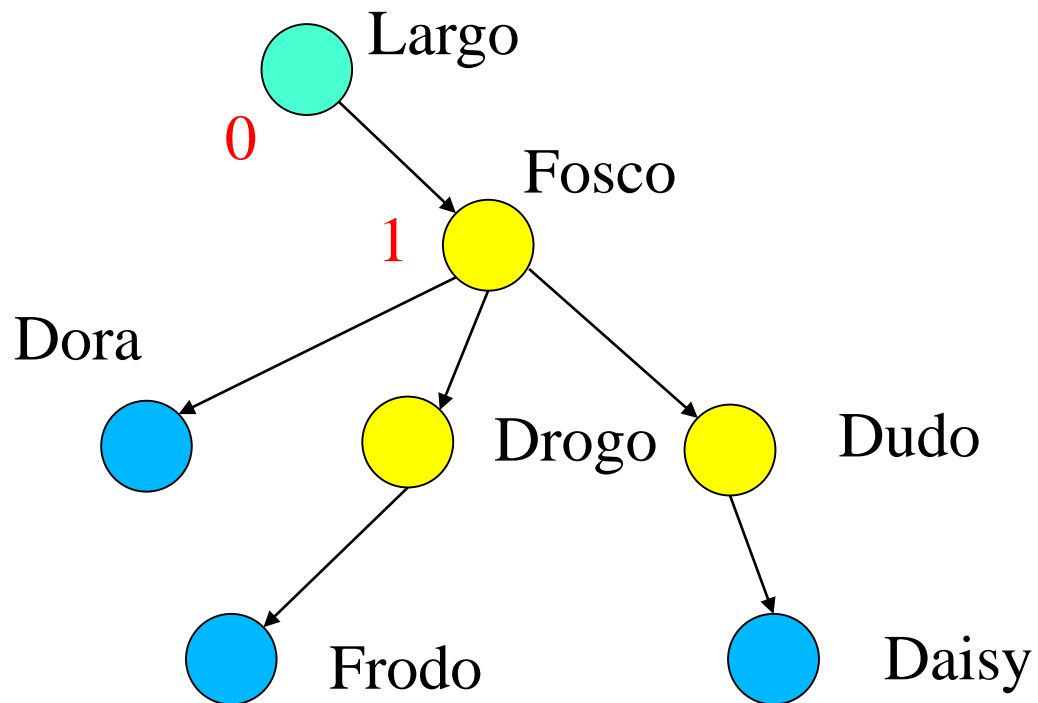
# Esempio

- Livelli



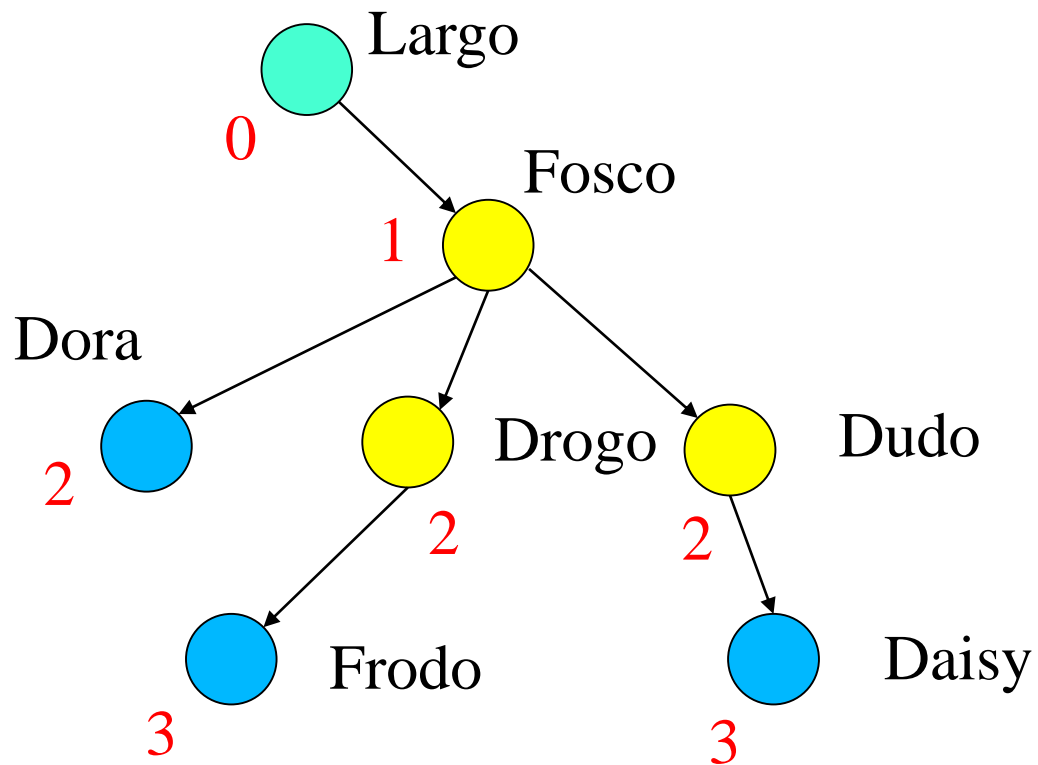
# Esempio

- Livelli



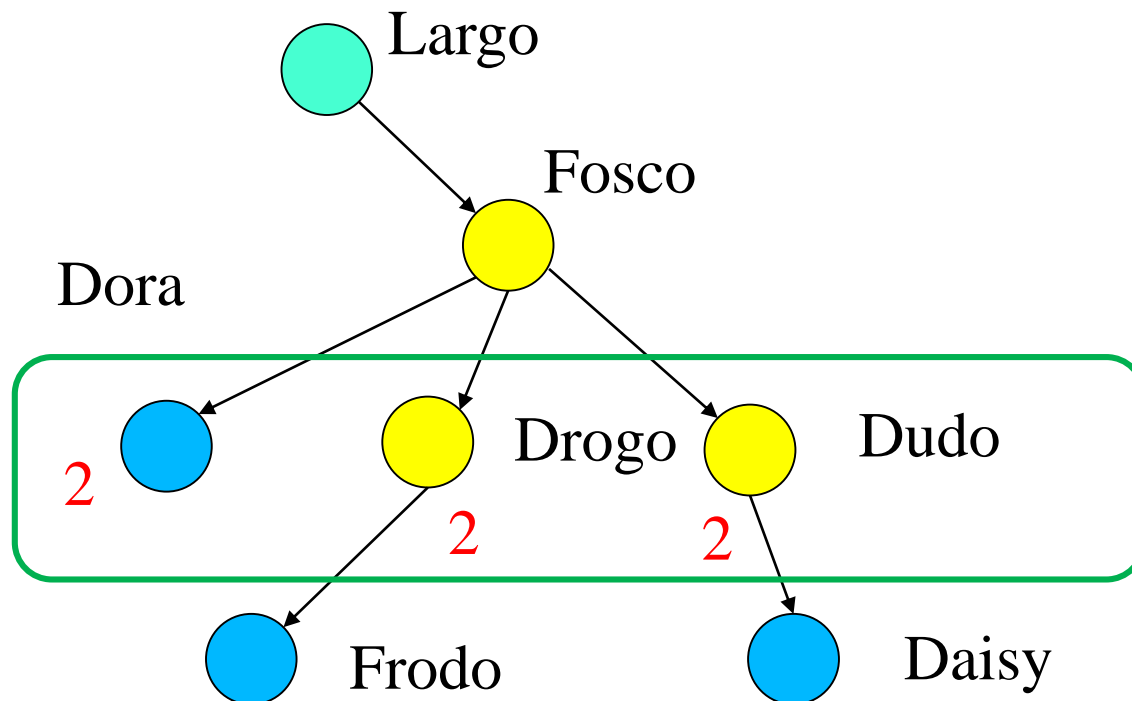
# Esempio

- Livelli



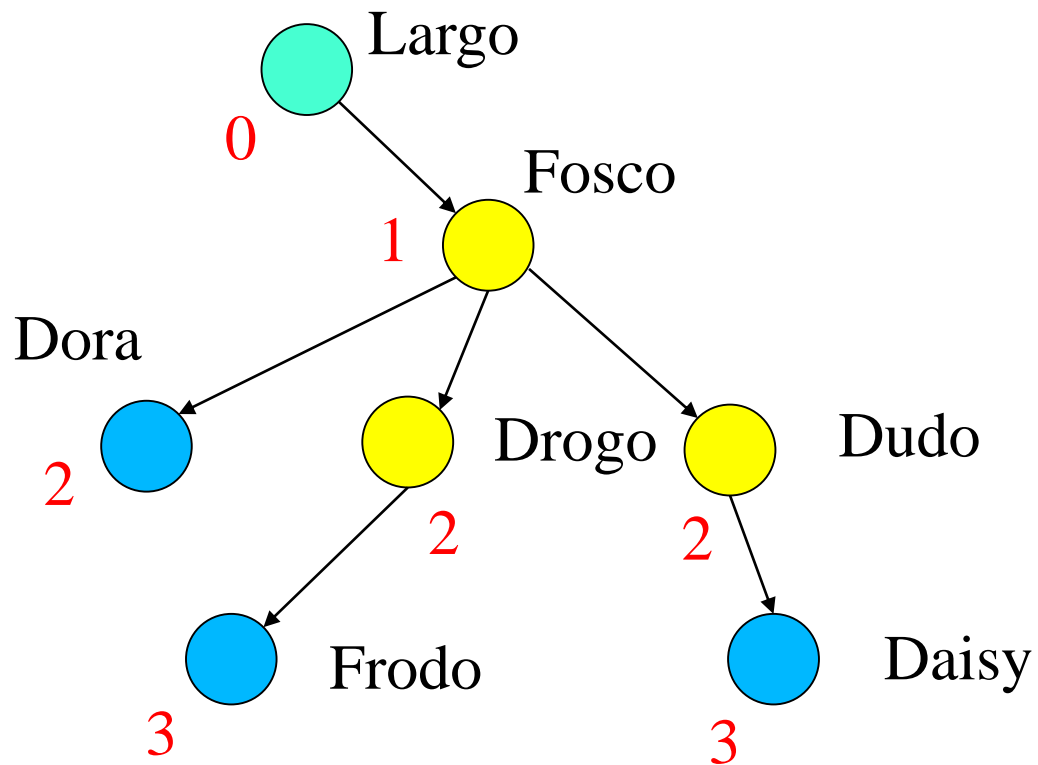
# Esempio

- Livello 2



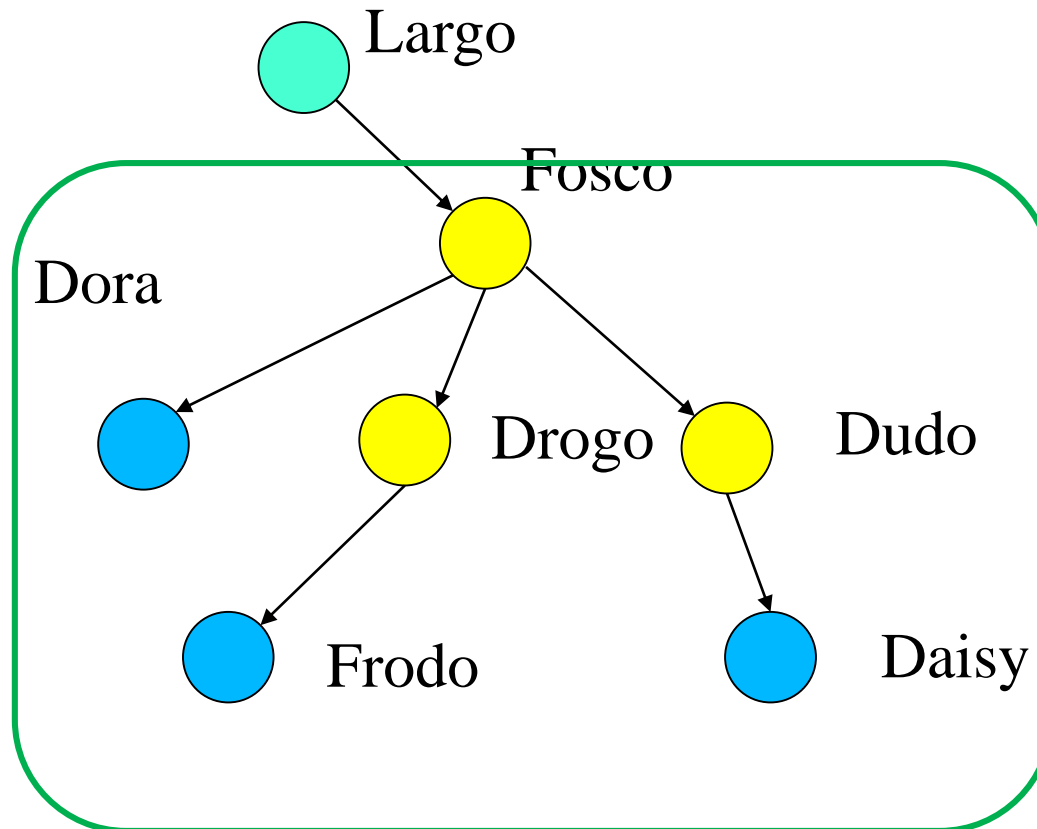
# Esempio

- L'albero ha altezza 3



# Esempio

- Ogni nodo può essere visto come la radice di un albero più piccolo (sottoalbero)

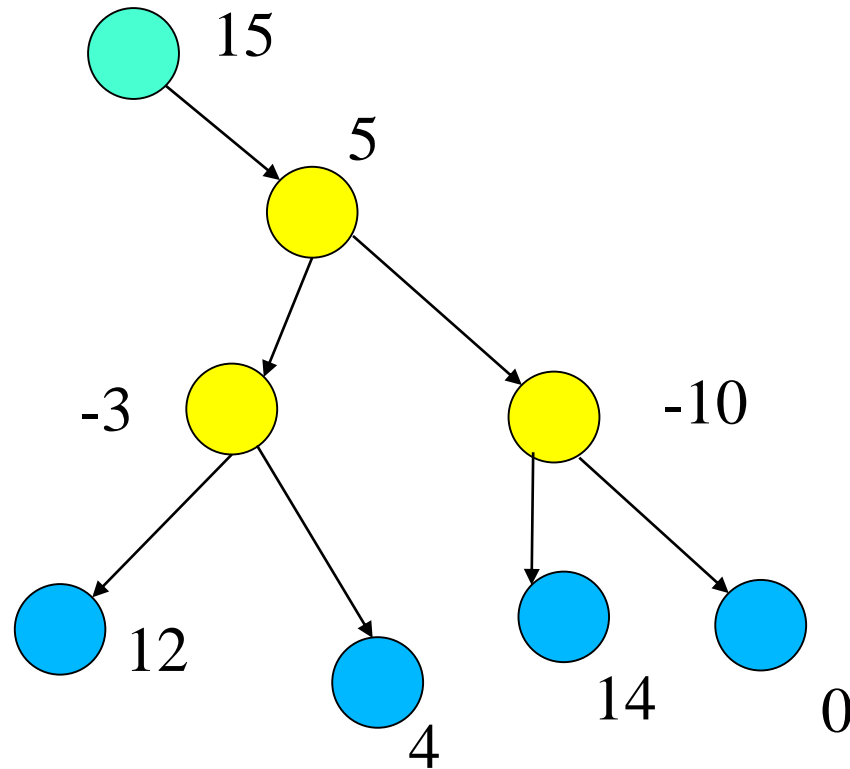


Sottoalbero di radice Fosco



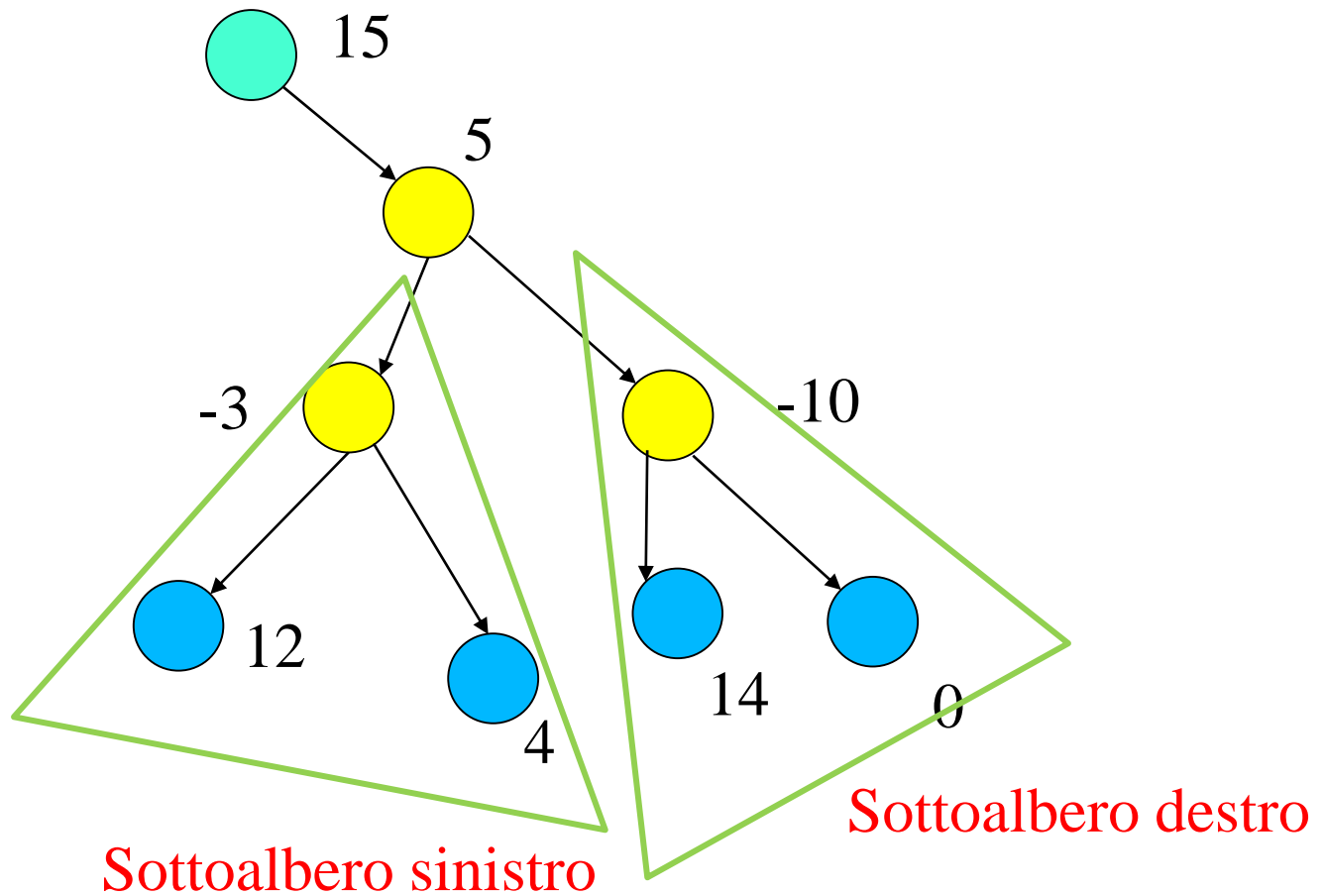
# Alberi binari

- Alberi con solo due figli
- Detti **figlio destro** e **figlio sinistro**
- Esempio:



# Alberi binari

- Alberi con solo due figli
  - Detti **figlio destro** e **figlio sinistro**
- Esempio:



# Alberi in C

- Come si realizza un albero binario in C ?
  - Ogni **nodo** è una struttura che contiene al suo interno due puntatori al suo stesso tipo, ad esempio se l'etichetta è un numero intero posso utilizzare:

```
typedef struct albero {  
    int val;  
    struct albero * left;  
    struct albero * right;  
} albero_t ;
```
  - Il puntatore **left** contiene il puntatore al figlio di sinistra mentre il puntatore **right** contiene il puntatore al figlio di destra
  - Le **foglie** utilizzano due puntatori nulli (**NULL**) nei campi **right left**

# Esempio: raddoppiare i pari

- Vogliamo raddoppiare solo le etichette pari
  - Lavorando con gli alberi è naturale usare funzioni ricorsive
  - Ragioniamo sul caso base:
    - Se l'albero è vuoto non devo fare niente
    - Se l'albero contiene solo un nodo foglia devo controllare se l'etichetta è pari e se lo è raddoppiare il valore
  - Supponiamo di saper risolvere il problema per un albero di livello  $n$ , allora per risolvere il problema per un albero di livello  $n+1$  posso:
    - Controllare l'etichetta della radice e raddoppiarla se è pari
    - Chiamare ricorsivamente la funzione che risolve il problema per un albero di livello  $n$  sia sul sottoalbero destro che sul sottoalbero sinistro

# Esempio: raddoppiare i pari

- Codifica

```
void raddoppia_pari(albero_t* r) {  
    if ( r == NULL ) return ;  
    if ( ( r -> val ) % 2 == 0 )  
        r -> val = 2 * ( r -> val ) ;  
    raddoppia_pari(r->left);  
    raddoppia_pari(r->right);  
}
```

# Osservazioni

- Nell'esempio precedente abbiamo scelto di operare analizzando nell'ordine
  - La radice dell'albero
  - Il sottoalbero di sinistra
  - Il sottoalbero di destra
- Poiche l'analisi dei sottoalberi sinistro e destro avviene utilizzando la stessa procedura ricorsiva, anche la loro analisi opera allo stesso modo
  - (prima la radice, poi il sottoalbero sx, quindi il sottoalbero dx . . . )

# Osservazioni

- Avremmo potuto procedere diversamente :

```
void raddoppia_pari(albero_t* r) {  
    if ( r == NULL ) return ;  
    raddoppia_pari(r->left);  
    if ( ( r -> val ) % 2 == 0 )  
        r -> val = 2 * ( r -> val ) ;  
    raddoppia_pari(r->right);  
}
```

- Oppure .....

# Osservazioni

- Oppure .....

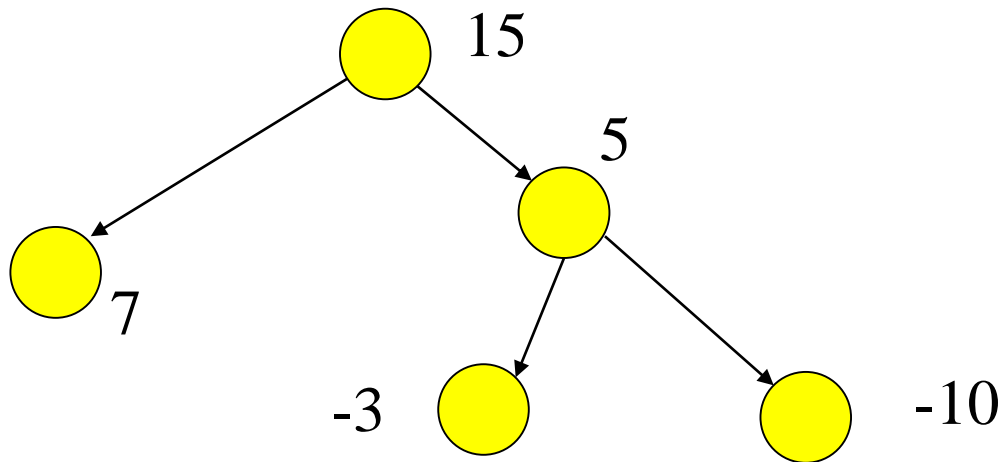
```
void raddoppia_pari(albero_t* r) {  
    if ( r == NULL ) return ;  
    raddoppia_pari(r->left);  
    raddoppia_pari(r->right);  
  
    if ( ( r -> val ) % 2 == 0 )  
        r -> val = 2 * ( r -> val ) ;  
}
```

- In questo caso è indifferente, perchè è indifferente l'ordine in cui elaboriamo i nodi dell'albero ....



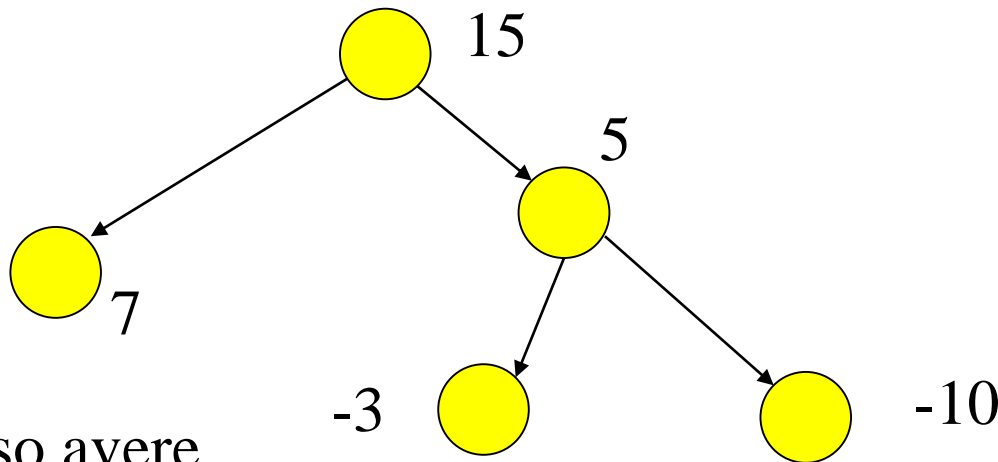
# Osservazioni

- Ci sono molti casi in cui l'ordine in cui vengono elaborati i nodi di un albero è rilevante e modifica il risultato
- Un esempio è la stampa delle etichette,



# Osservazioni

- Ci sono molti casi in cui l'ordine in cui vengono elaborati i nodi di un albero è rilevante e modifica il risultato
- Un esempio è la stampa delle etichette,



Posso avere

7, 15, -3, 5, -10 (prima il sottoalbero di sinistra)

-10, 5, -3, 15, 7 (prima il sottoalbero di destra)

15, 7, 5, -3, -10 (prima la radice, poi sx) .... etc

# Visita di un albero

- Visitare un albero significa **analizzare in sequenza tutti i suoi nodi**
- Molti problemi sugli alberi sono varianti del problema di effettuare una visita
- Ci sono diversi tipi di visita, che differiscono nell'ordine in cui vengono analizzati i nodi
- Visite **depth-first** (in profondità), sono di tre tipi :
  - **visita anticipata**: si analizza la radice, poi si effettua la visita anticipata del sottoalbero SX e poi si effettua la visita anticipata del sottoalbero DX
  - **visita simmetrica**: si effettua la visita simmetrica del sottoalbero SX, poi si analizza la radice e poi si effettua la visita simmetrica del sottoalbero DX
  - **visita posticipata**: si effettua la visita posticipata del sottoalbero SX, poi si effettua la visita posticipata del sottoalbero DX, e infine si analizza la radice

# Visita di un albero

- Visitare un albero significa **analizzare in sequenza tutti i suoi nodi**
- Molti problemi sugli alberi sono varianti del problema di effettuare una visita
- Ci sono diversi tipi di visita, che differiscono nell'ordine in cui vengono analizzati i nodi
  - Visite **depth-first** (in profondità), sono di tre tipi :
    - **visita anticipata**
    - **visita simmetrica**
    - **visita posticipata**
  - Visita **breadth-first** (per livelli): si visita prima la radice (livello 0), poi si visitano tutti i nodi di livello 1, poi tutti i nodi di livello 2, etc ....

# Visita simmetrica

- Supponiamo di avere una funzione

```
void analizza_nodo (albero_t* n) ;
```

Questa funzione elabora il valore del singolo nodo, secondo il problema che dobbiamo risolvere, eventualmente modificandolo

- Quindi la funzione che effettua la visita simmetrica è

```
void visitaSimmetrica (albero_t* bt) {  
    if (bt == NULL) return;  
    visitaSimmetrica (bt -> left) ;  
    analizza_nodo (bt) ;  
    visitaSimmetrica (bt -> right) ;  
}
```

# Visita anticipata

```
void visitaAnticipata (albero_t* bt) {  
    if (bt == NULL) return;  
    analizza_nodo(bt);  
    visitaAnticipata(bt -> left);  
    visitaAnticipata(bt -> right);  
}
```

# Visita posticipata

```
void visitaPosticipata (albero_t* bt) {  
    if (bt == NULL) return;  
    visitaPisticipata (bt -> left) ;  
    visitaPosticipata (bt -> right) ;  
    analizza_nodo (bt) ;  
}
```

# Esempio : ricerca

- Supponiamo di avere un tipo albero in cui l'etichetta dei nodi è una stringa
- Scrivere una funzione di prototipo

```
int ricerca (albero_t* bt, char * s);
```

che restituisce 1 se esiste la stringa **s** nell'albero **bt** e 0 altrimenti



# Esempio : ricerca

- Possiamo modificare una qualsiasi delle tre visite viste.
  - Scegliamo la visita anticipata,
  - Nel nostro caso l'analisi del nodo richiede il confronto di due stringhe
  - Inoltre bisogna riportare correttamente il valore restituito dalle chiamate ricorsive alla funzione

# Esempio : ricerca

```
int ricerca (albero_t* bt, char* s) {  
    if (bt == NULL) return 0;  
    if (strcmp(bt->val,s) == 0 ) return 1;  
    if ( ricerca(bt -> left,s) == 1 )  
        return 1;  
    return ricerca(bt -> right,s);  
}
```

# Esempio : somma

- Supponiamo di avere un tipo albero in cui l'etichetta dei nodi è un double
- Scrivere una funzione di prototipo

**double somma (albero\_t\* bt) ;**

che restituisce la somma di tutte le etichette presenti nell'albero

# Esempio : somma

- Possiamo modificare una qualsiasi delle tre visite viste.
  - Scegliamo la visita posticipata
  - Nel nostro caso l'analisi del nodo richiede la somma con i valori ottenuti dal sottoalbero di destra e di sinistra

# Esempio : somma

```
double somma_alb (albero_t* bt) {  
    double sx, dx;  
    if (bt == NULL) return 0;  
    sx = somma_alb(bt -> left);  
    dx = somma_alb(bt -> right);  
    return sx + bt->val + dx;  
}
```

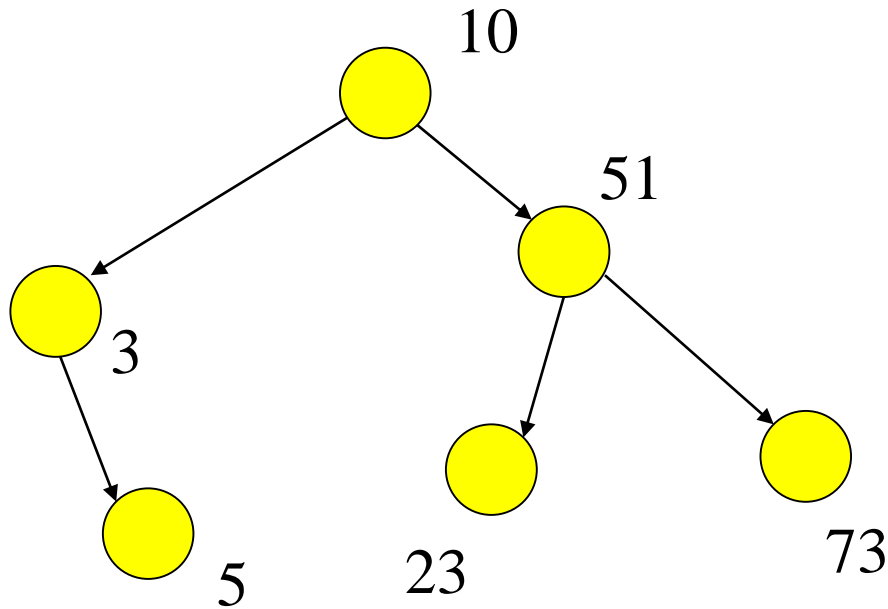
# Esempio : alberi binari di ricerca

- Supponiamo di avere un albero binario tale che sulle etichette dei nodi è definito un ordinamento totale
  - Interi, reali, stringhe
  - Indichiamo l'operatore con  $\leq$
- Un albero binario di ricerca è un albero in cui
  - Per ogni nodo  $n$ , indicati con  $s(n)$  e  $d(n)$  il figlio di sinistra e destra, le etichette verificano la seguente relazione

$$E(s(n)) \leq E(n) \leq E(d(n))$$

# Ad esempio

- Etichette intere, operatore minore o uguale



# Esercizi

- Supponiamo di avere un albero binario di ricerca con etichette intere
  - Scrivere una funzione che cerca se un valore  $x$  è presente nell'albero con un numero di passi pari all'altezza dell'albero
  - Scrivere una funzione che inserisce una nuova etichetta  $x$  nell'albero mantenendo l'ordinamento fra i nodi
  - Scrivere una funzione che cancella un nodo dell'albero con una certa etichetta  $x$  mantenendo l'albero ordinato