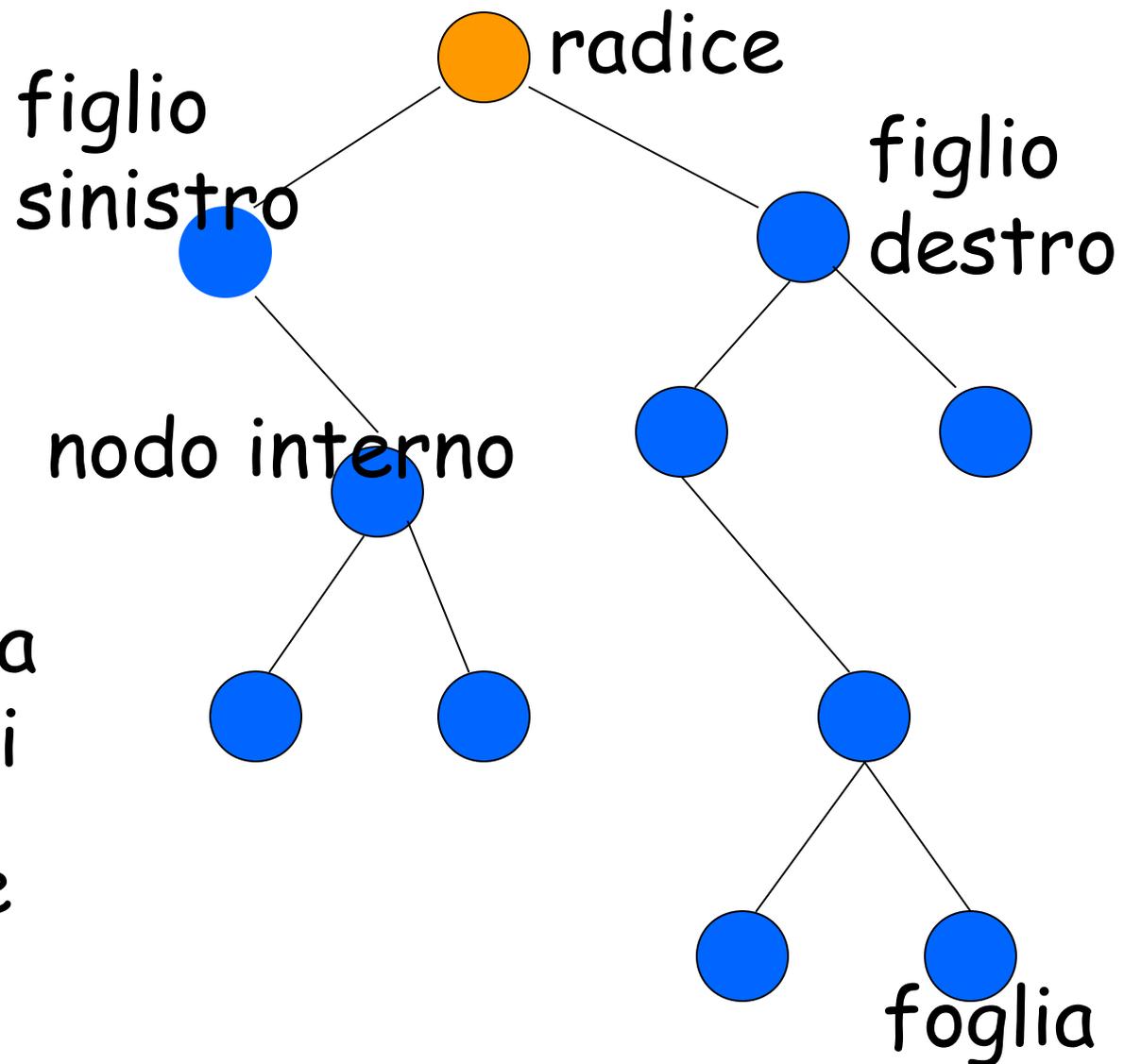


alberi binari e
ricorsione

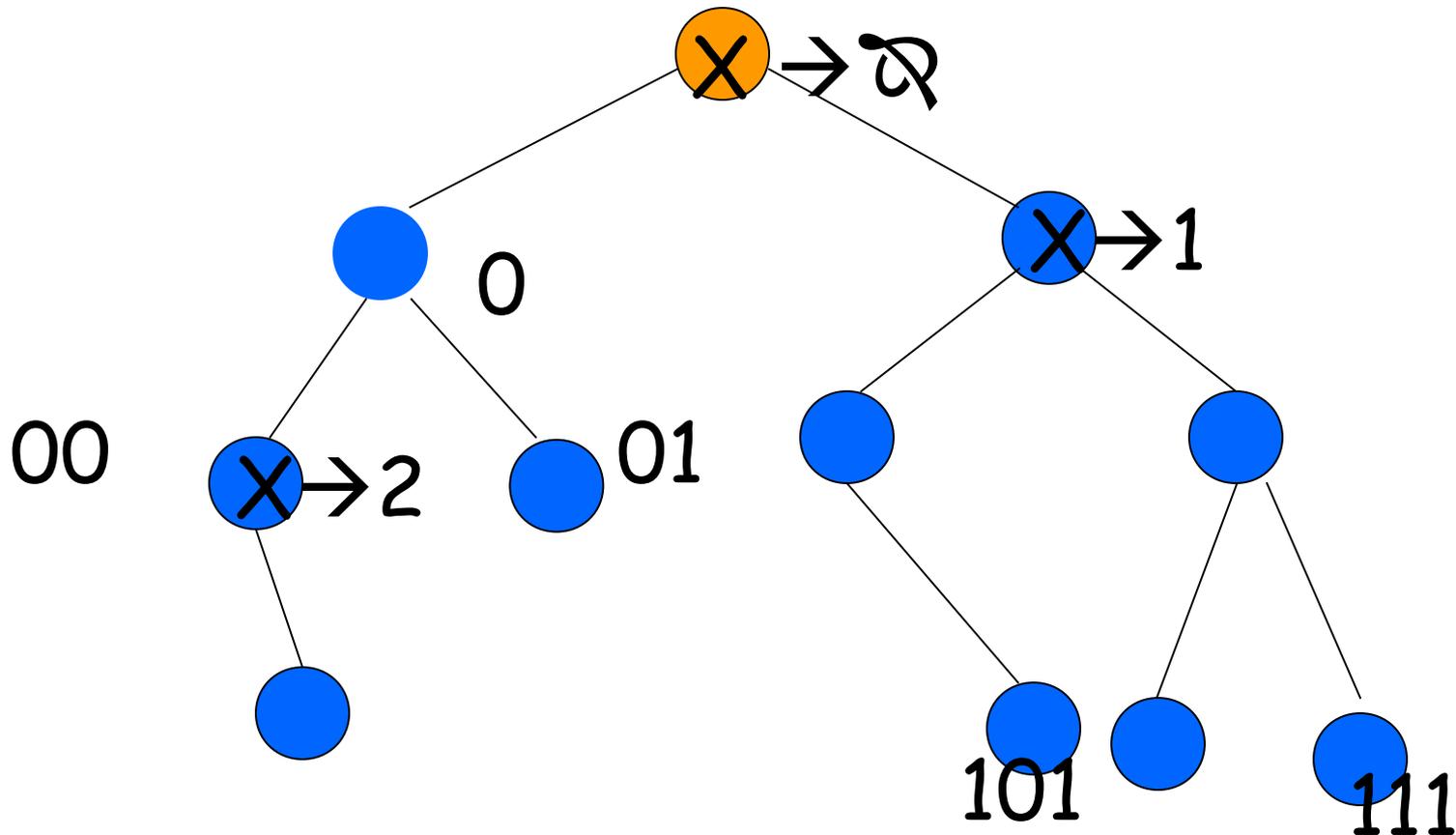
un albero binario:



ogni nodo ha
al più 2 figli

ogni figlio è
destro o
sinistro

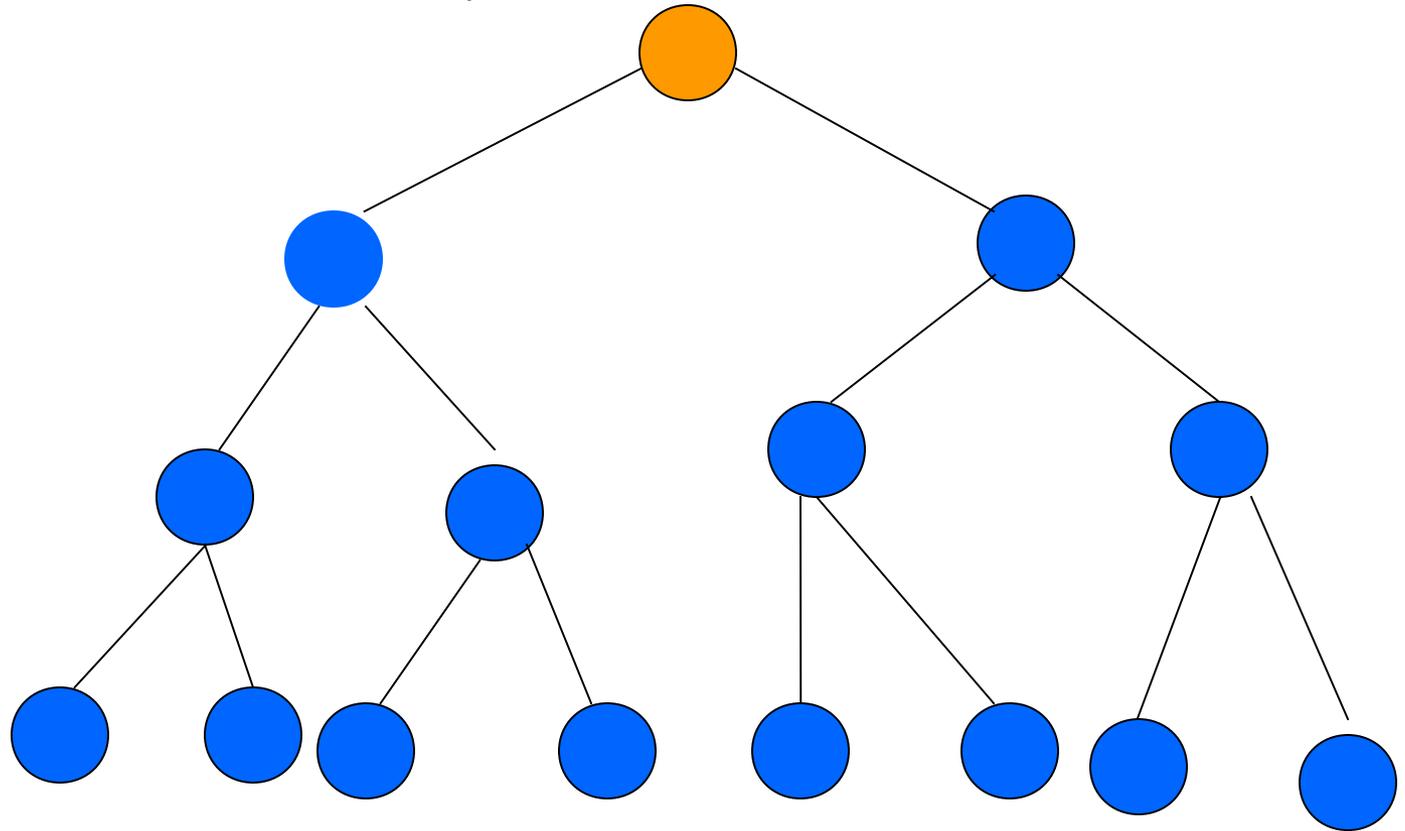
cammini = sequenze di nodi = sequenze di 0 e 1



profondità di un nodo

altezza dell'albero=prof. max delle foglie

albero binario completo



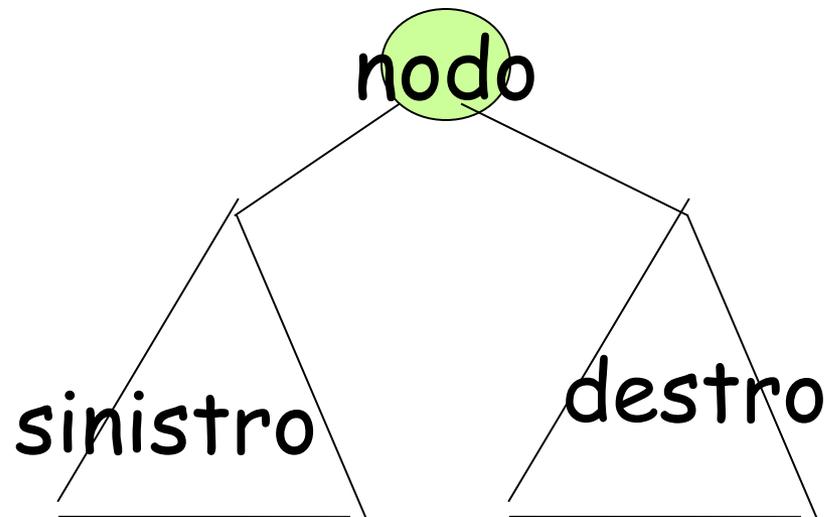
ogni livello è completo, se h = altezza

l'albero contiene $2^{h+1} - 1$ nodi

definizione **ricorsiva** degli alberi:

albero binario è:

- un albero vuoto
- **nodo(albero sinistro, albero destro)**



attraversamenti degli alberi = modi di visitare tutti i loro nodi

in profondità = depth-first

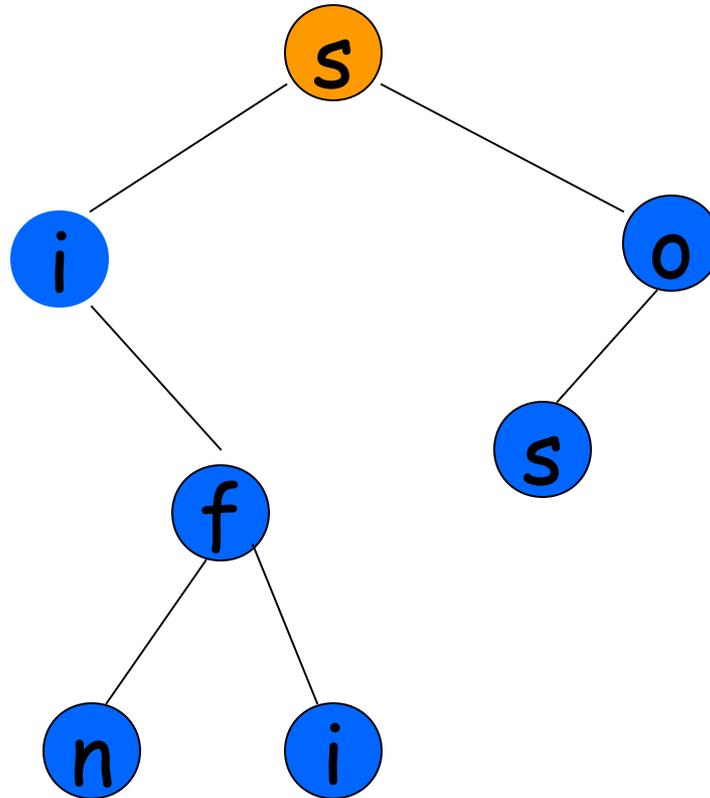
ma anche in larghezza = breath-first

percorso **infisso**:

1. a sinistra

2. nodo

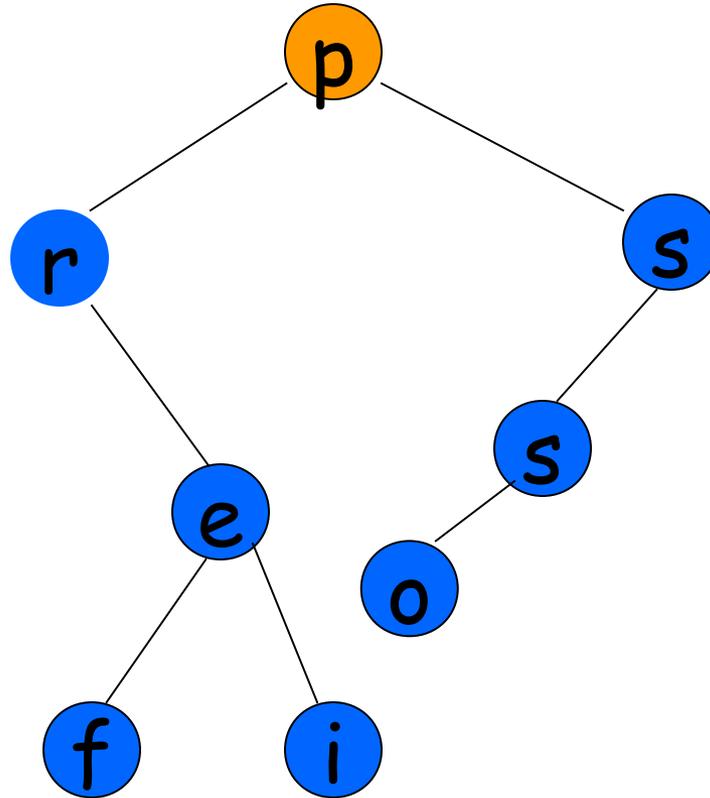
3. a destra



in profondità da sinistra a destra

percorso **prefisso**:

1. nodo
2. a sinistra
3. a destra

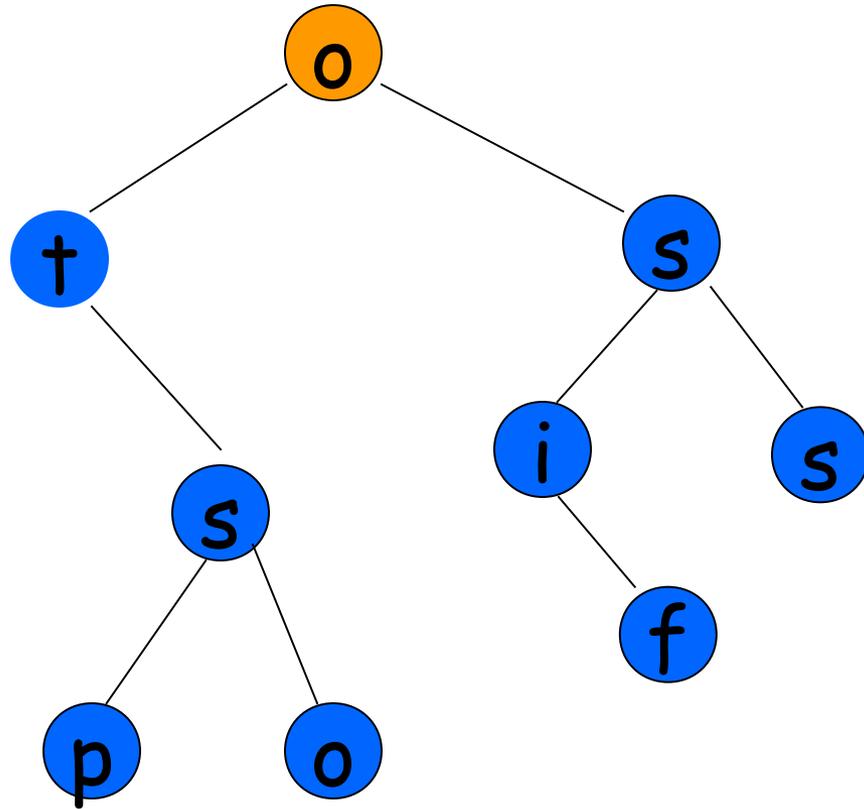


percorso **postfisso**:

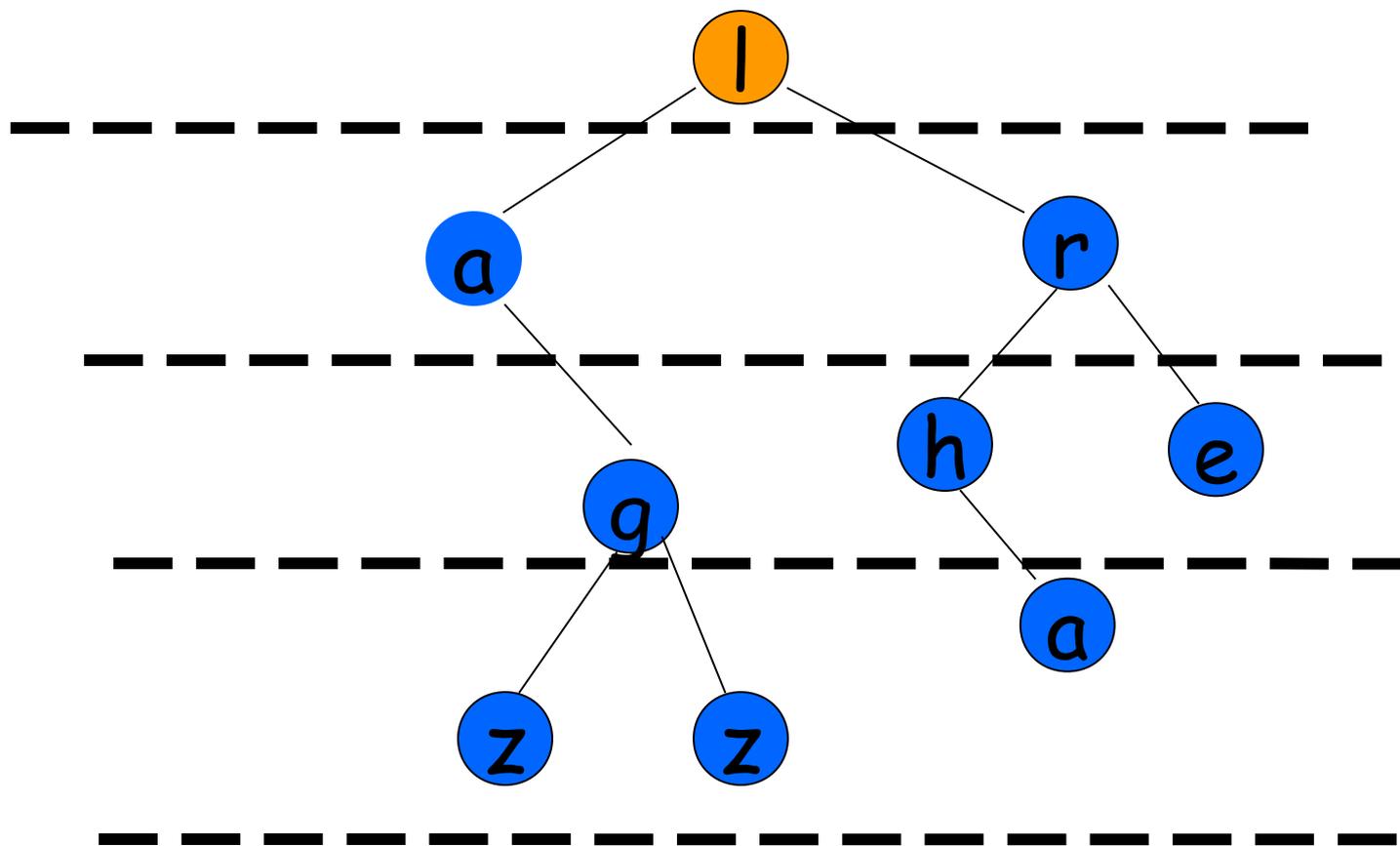
1. a sinistra

2. a destra

3. nodo



in larghezza



come realizzare un nodo di un albero binario in Python:

```
class Albero:
    def __init__(self, val=None, sx=None, dx=None):
self.val = val
        self.sx = sx
        self.dx = dx
```

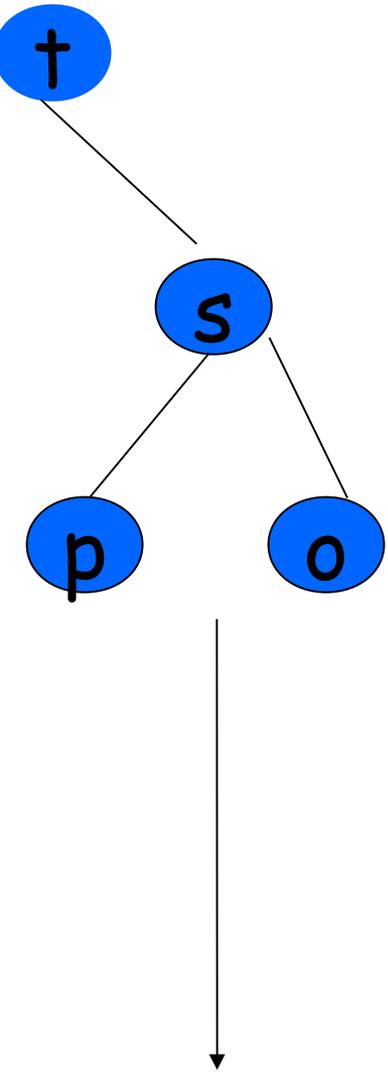
costruiamo questo albero:

```
P = Albero('p')
```

```
O = Albero('o')
```

```
S = Albero('s', P, O)
```

```
T = Albero('t', None, S)
```



`t(_,s(p(_,_),o(_,_)))` rappresentazione lineare

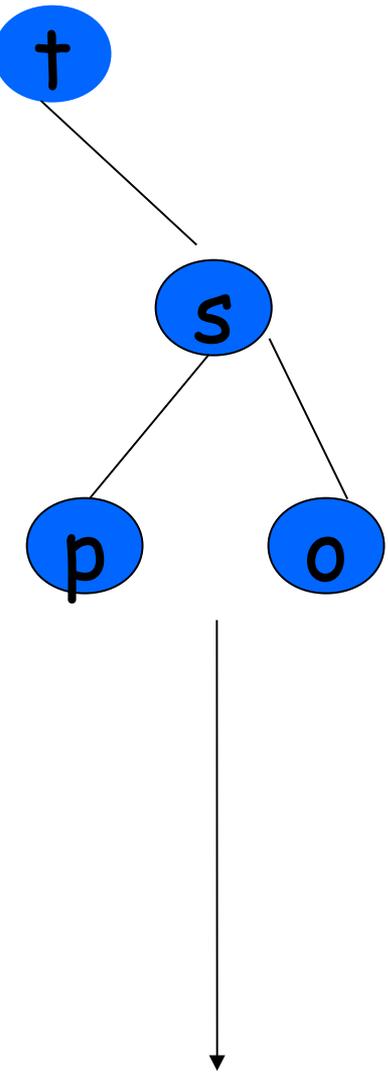
costruiamo questo albero:

```
P = Albero('p')
```

```
O = Albero('o')
```

```
S = Albero('s', P, O)
```

```
T = Albero('t', None, S)
```

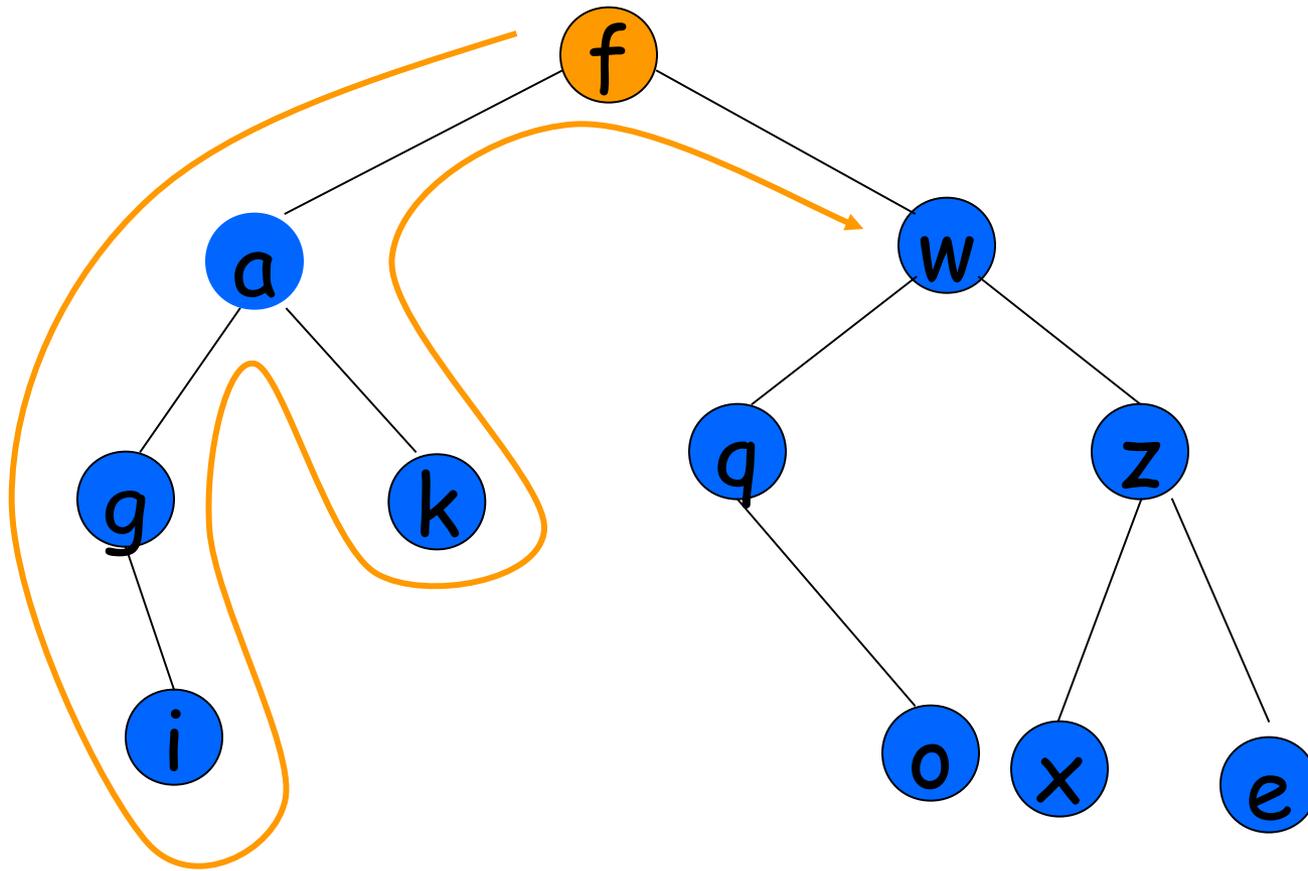


`t(_,s(p(_,_),o(_,_)))` rappresentazione lineare

Operazioni su alberi binari

- 1) Stampa lineare di un albero
- 2) Trova un elemento nell'albero

Vediamone altre...

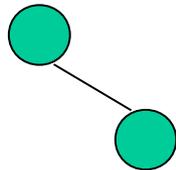


cerchiamo 'w'

f -> fa -> fag -> fagi -> fag -> fa -> fak ->
fa -> f -> fw

altezza di un albero = profondità massima
dei suoi nodi = distanza massima tra nodi
e "antenati" nell'albero

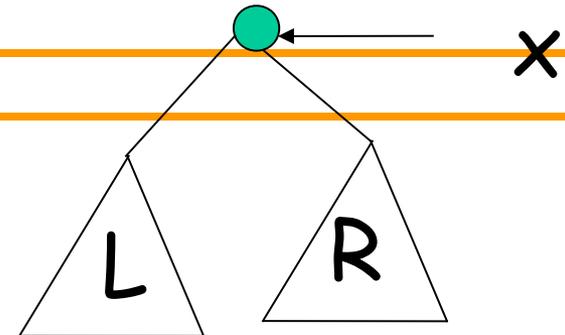
● altezza 0

 ● ● altezza 1

albero vuoto? per convenzione -1

calcolo ricorsivo dell'altezza:

```
def altezza(albero):  
    if not albero:  
        return -1  
  
    return max(altezza(albero.sx), altezza(albero.dx)) + 1;
```



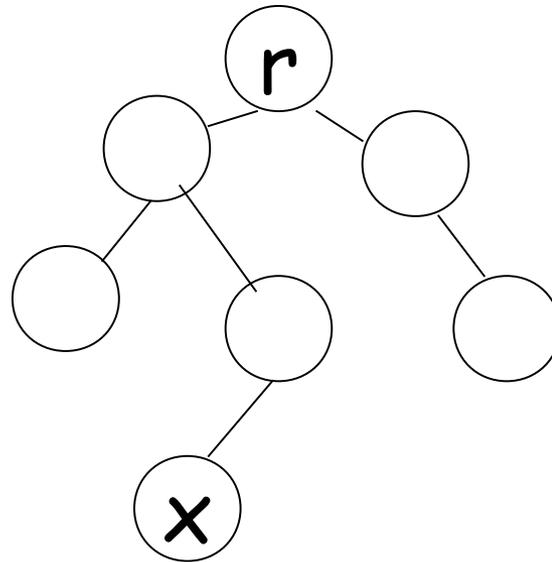
Procedura corretta perché:

1) albero vuoto ok!

2) se non vuoto, assumiamo ok per alberi L e R e lo dimostriamo induttivamente

un **cammino** di un albero = sequenza di 0 e 1

0=sinistra 1= destra



cammino per x:

$C=[0,1,0]$ lung=3

cammino di r $C=[]$ rlung =0

Operazioni su alberi binari

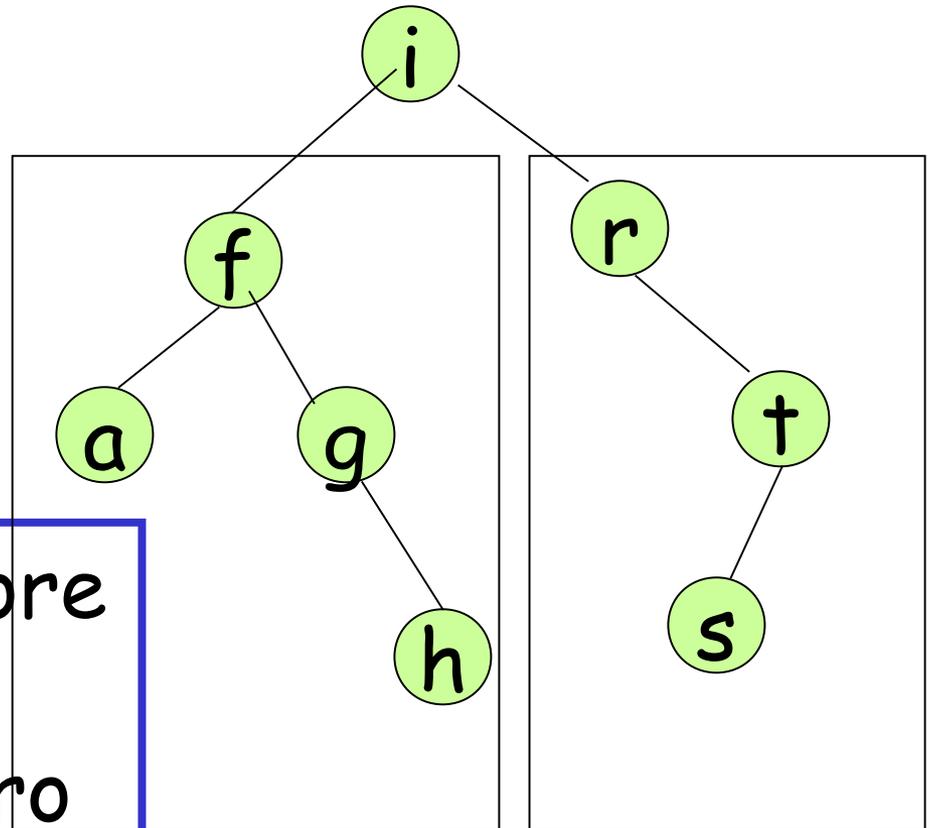
- 1) Stampa lineare di un albero
- 2) Trova un elemento nell'albero
- 3) Dato un cammino C , restituire l'albero radicato nel nodo corrispondente

Data una lista C che contiene un cammino, restituire il nodo corrispondente

```
def trova_cammino(albero, C):  
    if not C:  
        return albero  
  
    if not albero:  
        return None  
  
    if C[0]==0:  
        return trova_cammino(albero.sx, C[1:])  
  
    else:  
        return trova_cammino(albero.dx, C[1:])
```

binary search trees (BST):

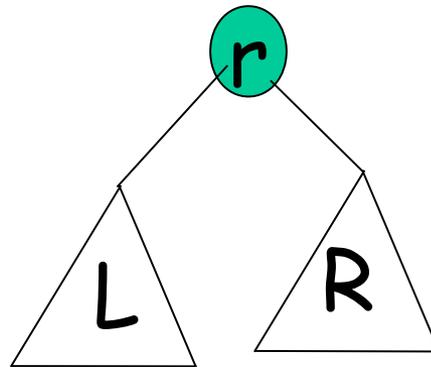
Detti anche
alberi ordinati



ogni nodo è maggiore
dei nodi nel suo
sottoalbero sinistro
e minore di quelli del
suo sottoalbero
destro

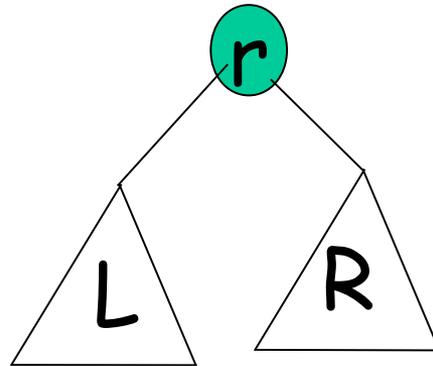
in un BST è **facile** (efficiente) trovare un nodo con un certo campo valore y e restituire quel nodo se lo troviamo e None altrimenti

in generale:



controllo r, cerco in L e se no in R o viceversa insomma se non c'è devo visitare tutti i nodi !!

in un BST la cosa è + semplice:



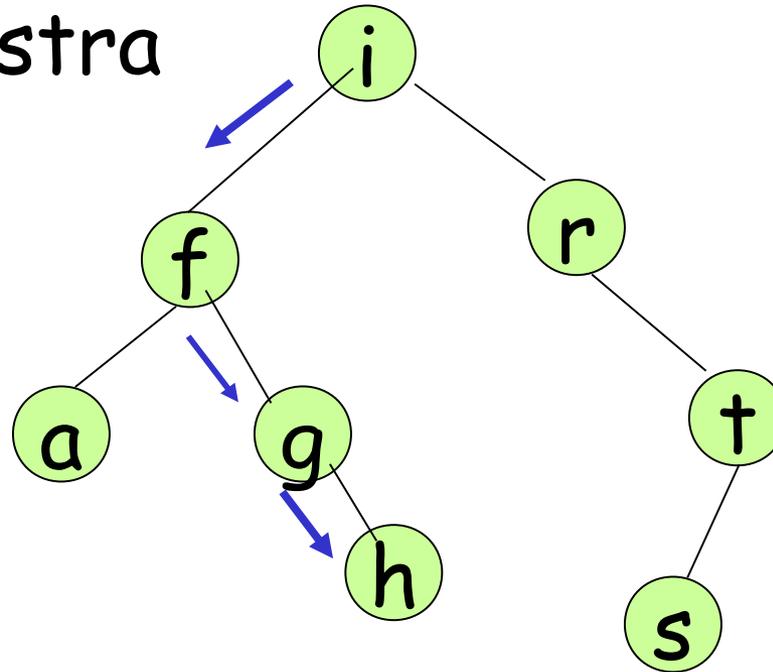
1. $r.val == y$ restituisco r , altrimenti:
2. se $r.val > y$ → cerco solo in L
altrimenti cerco solo in R

cerchiamo h:

$h < i$ andiamo a sinistra

$h > f$ destra

$h > g$ destra



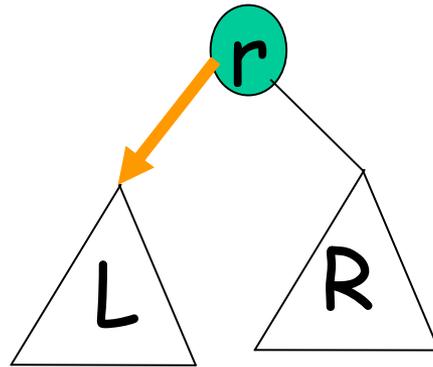
trovato !!!

ricerca in un BST:

```
def bst_search(albero, v):  
    if not albero:  
        return None  
  
    if albero.val==v:  
        return albero  
  
    if albero.val<v:  
        return bst_search(albero.sx,v)  
  
    return bst_search(albero.dx,v)
```

quante chiamate ricorsive si fanno al massimo?

seguo un solo cammino : al massimo farò tante invocazioni quant'è l'altezza dell'albero

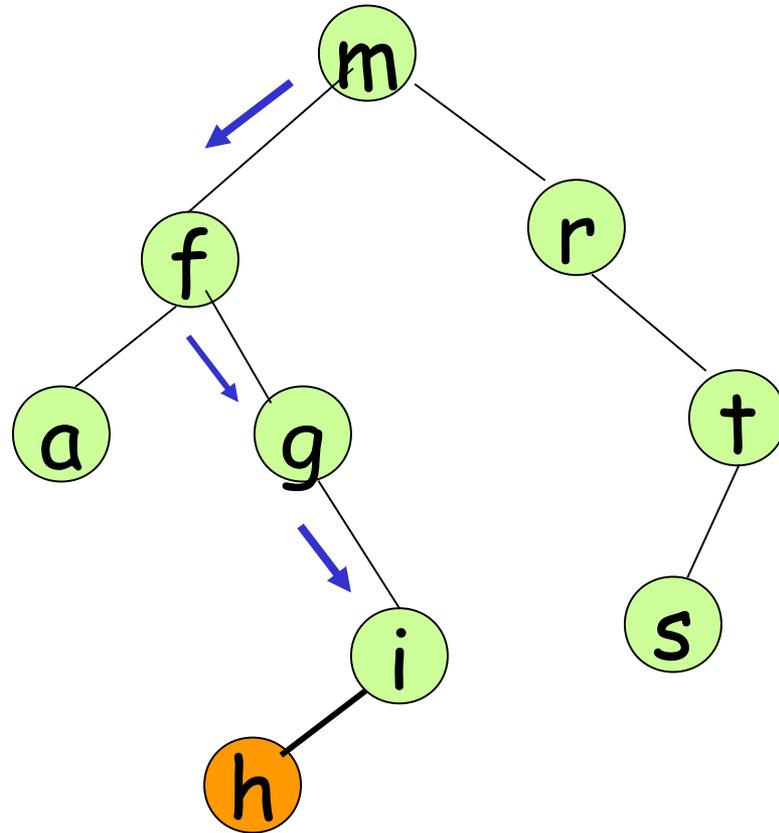


se l'albero è equilibrato, $altezza = \log n$

dove n è il numero dei nodi dell'albero

una bella differenza tra n e $\log n$!!!

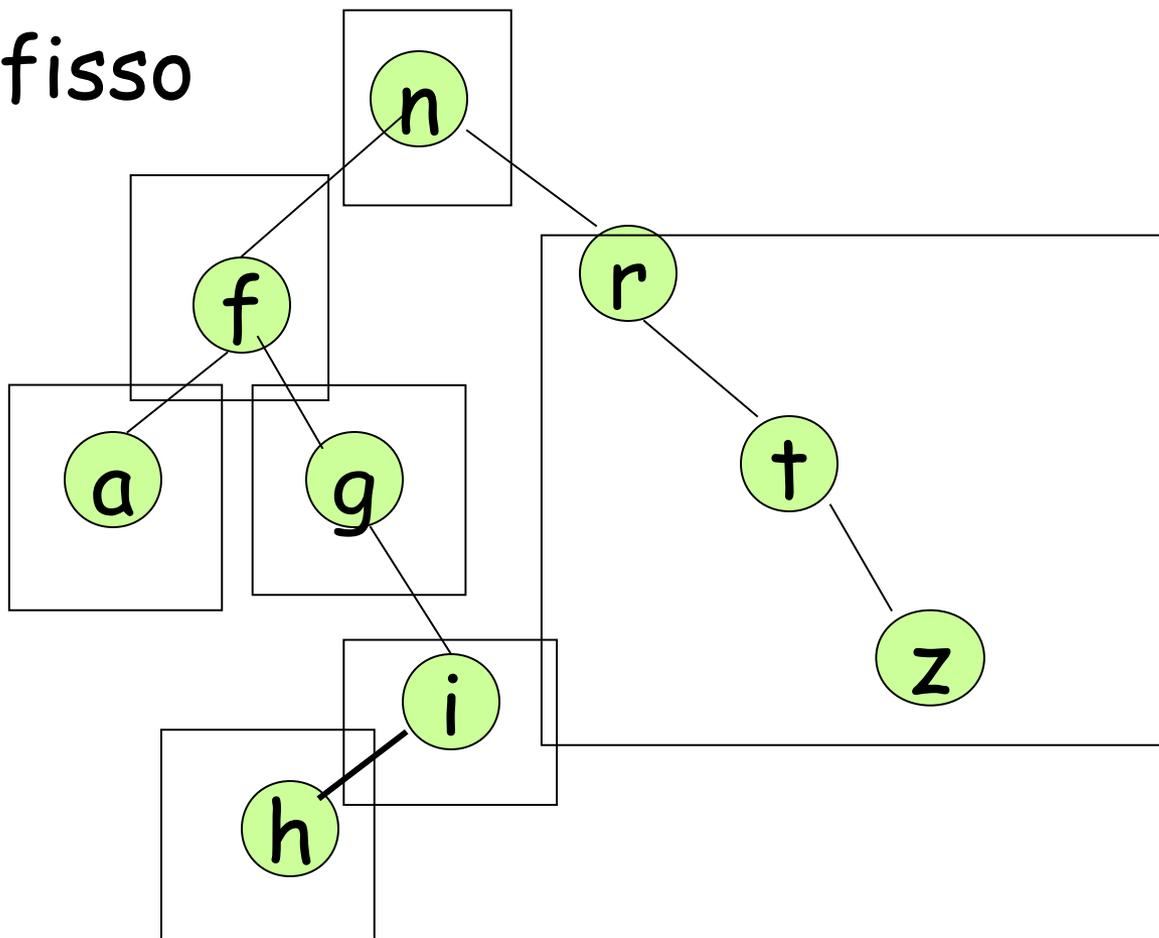
se h non ci fosse dove andrebbe inserito?



inserimento in un BST:

```
def bst_ins(albero, v):  
    if not albero:  
        return Albero(v)  
  
    if albero.val > v:  
        albero.sx = bst_ins(albero.sx, v)  
  
    if albero.val <= v:  
        albero.dx = bst_ins(albero.dx, v)  
  
    return albero
```

percorso infisso



a f g h i n r t z

sono in ordine !!

inseriamo



partendo dall'albero vuoto:

Verifica di un BST:

```
def is_bst(albero):  
    def lista_ordinata(lista):  
        if len(lista) <= 1:  
            return True  
        for i in range(len(lista)-1):  
            if lista[i] > lista[i+1]:  
                return False  
        return True  
    v = visita_in(albero)  
    return lista_ordinata(v)
```

esercizi sugli alberi binari

1. calcolare num occorrenze di y
2. contare i nodi con esattamente 1 figlio
2. restituire una foglia
3. restituire un nodo di profondità k
4. stampare in ordine infisso i primi k nodi
5. restituire foglia a prof. minima

contare i nodi con esattamente un figlio

```
def cn_un_figlio(albero):  
    def unf(albero):  
        return albero.sx and not albero.dx or \  
            albero.dx and not albero.sx  
    if not albero:  
        return 0  
    cn = cn_un_figlio(albero.sx)+cn_un_figlio(albero.dx)  
    return 1+cn if unf(albero) else cn
```

1. come riconoscere un nodo di profondità k ?

parto dalla radice con k e lo diminuisco ad ogni livello finchè non diventa 0

quale cammino seguuo?

è arbitrario purchè si sia in grado di percorrerli tutti

non appena troviamo un nodo a profondità k , interrompiamo la ricorsione e ritorniamo

```
def at_prof(albero, k):  
    if not albero:  
        return None  
  
    if k==0:  
        return albero  
  
    p = at_prof(albero.sx, k-1):  
    return p if p else at_prof(albero.dx, k-1)
```