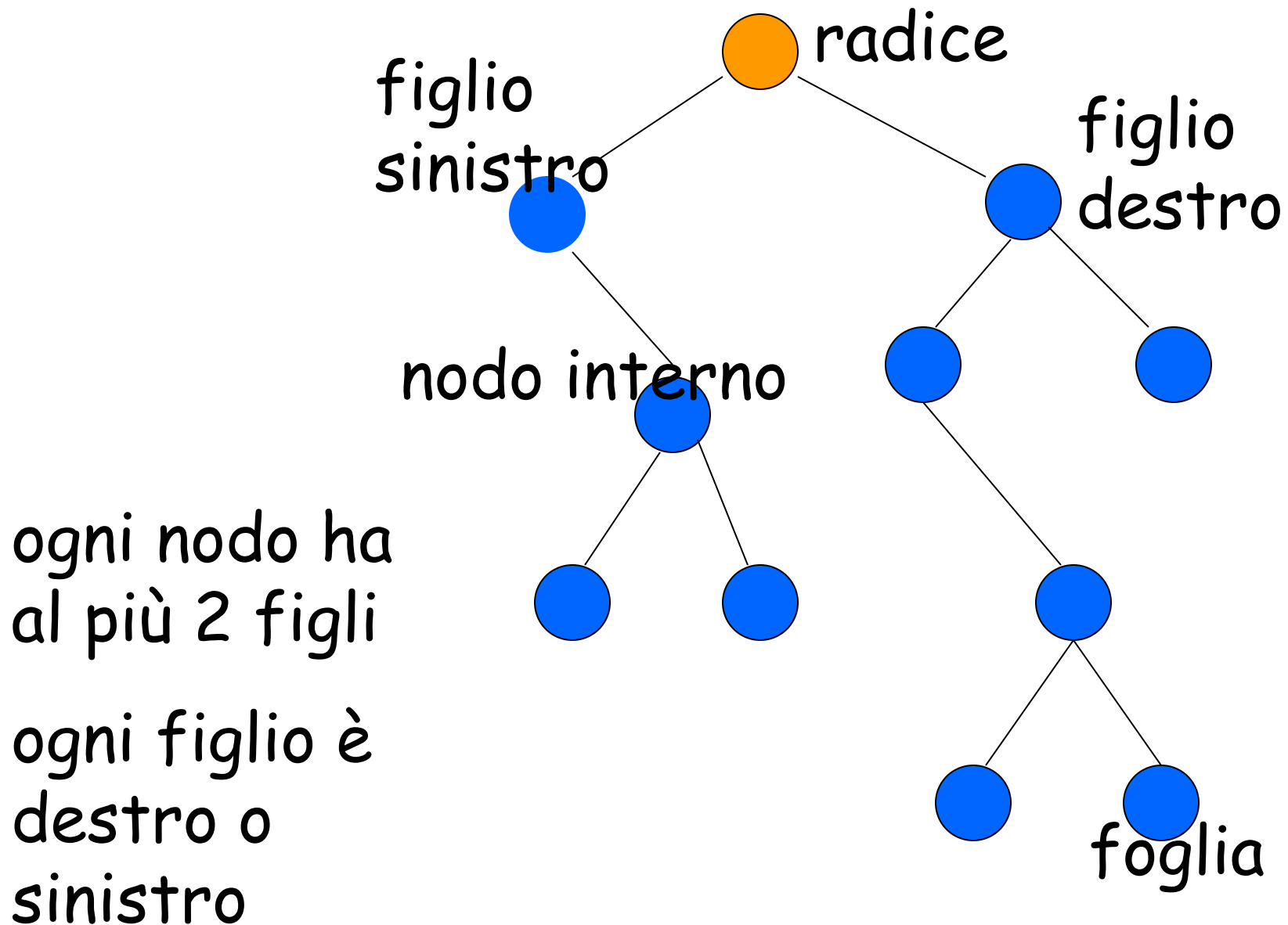


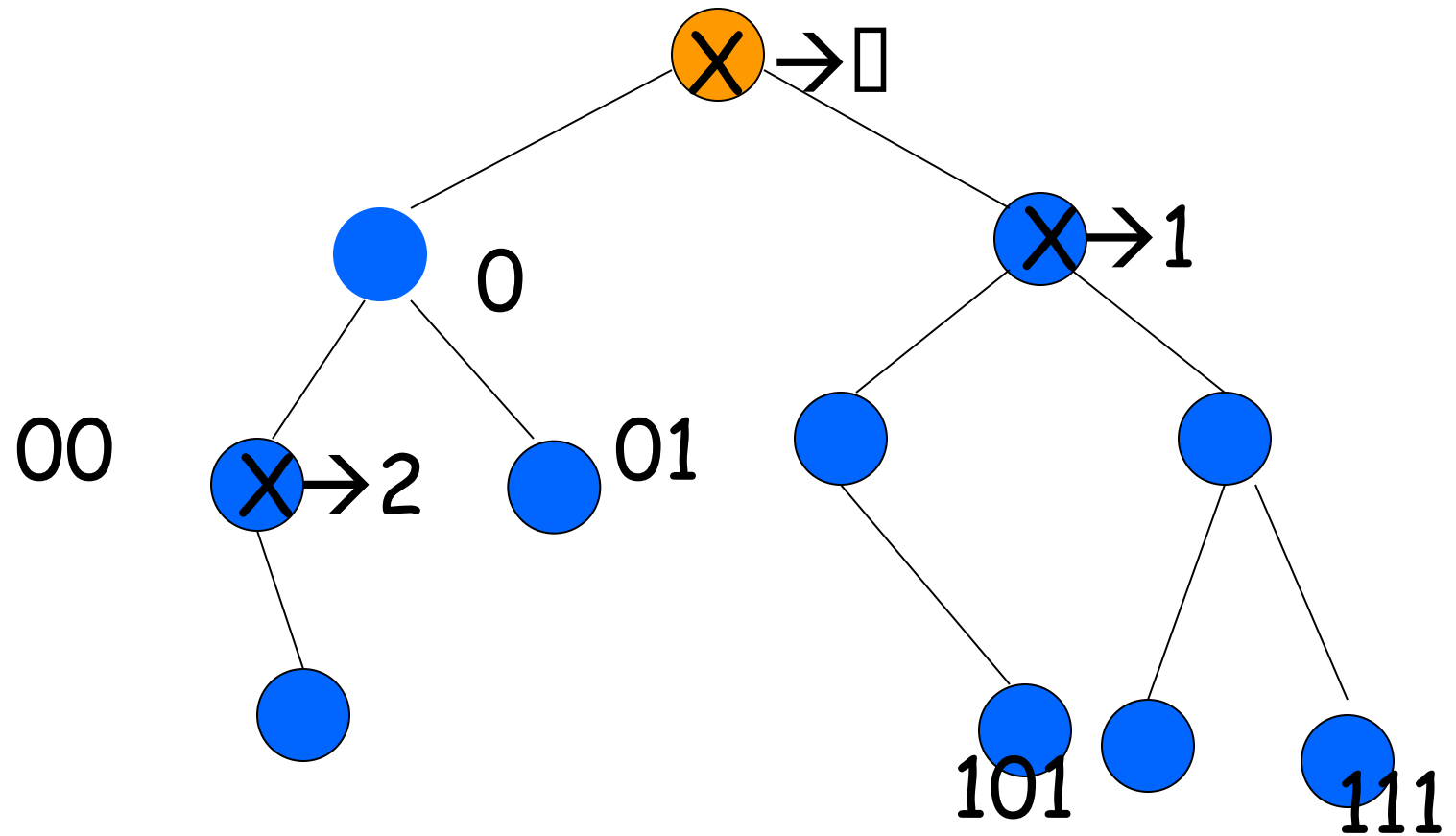
alberi binari e ricorsione

cap. 12

un albero binario:



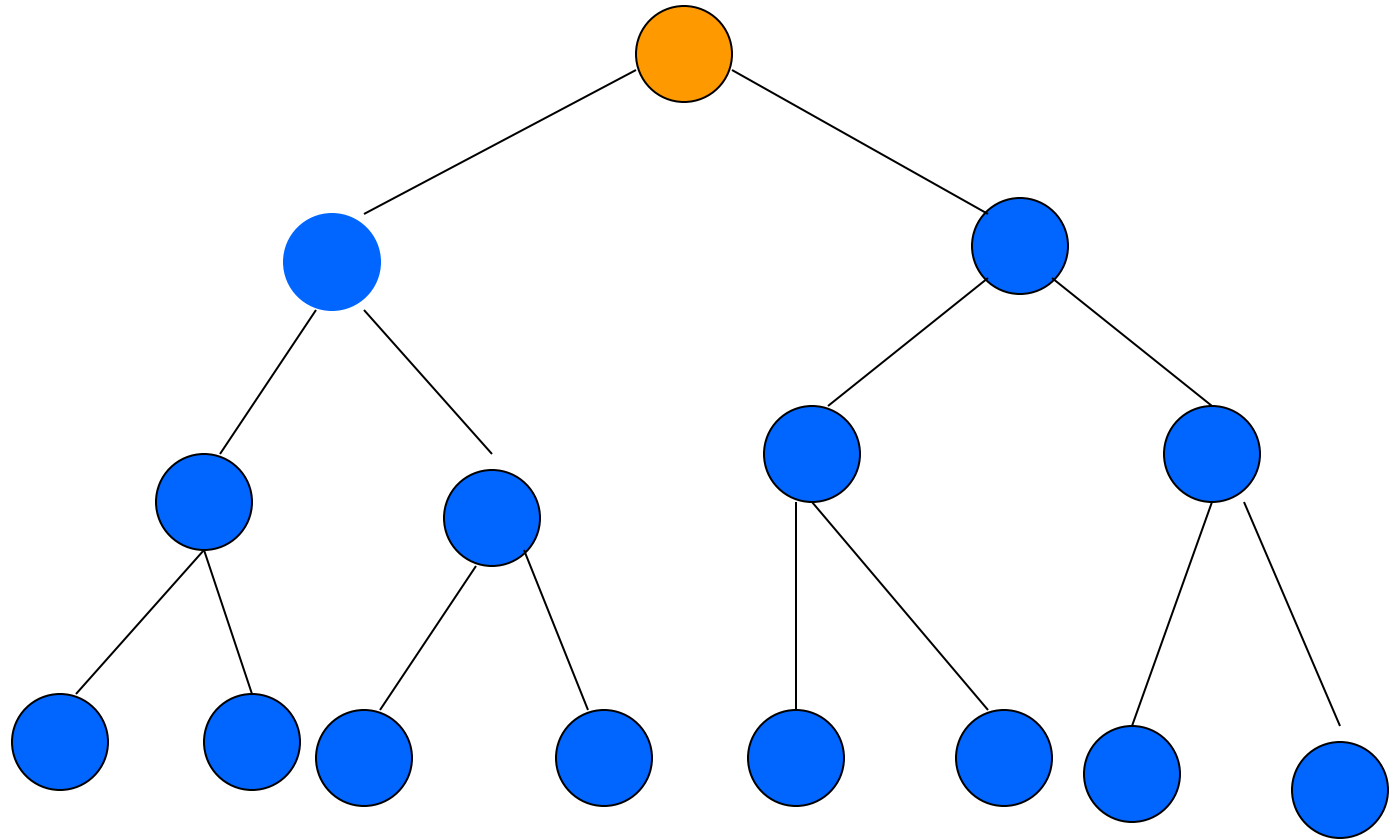
cammini = sequenze di nodi = sequenze di 0 e 1



profondità di un nodo

altezza dell'albero=prof. max delle foglie

albero binario completo

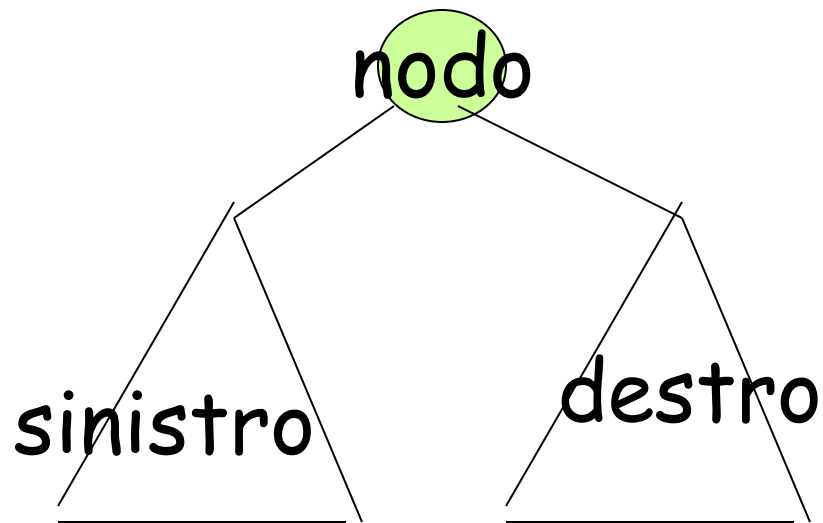


ogni livello è completo, se h = altezza
l'albero contiene $2^{h+1} - 1$ nodi

definizione **ricorsiva** degli alberi:

albero binario è:

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



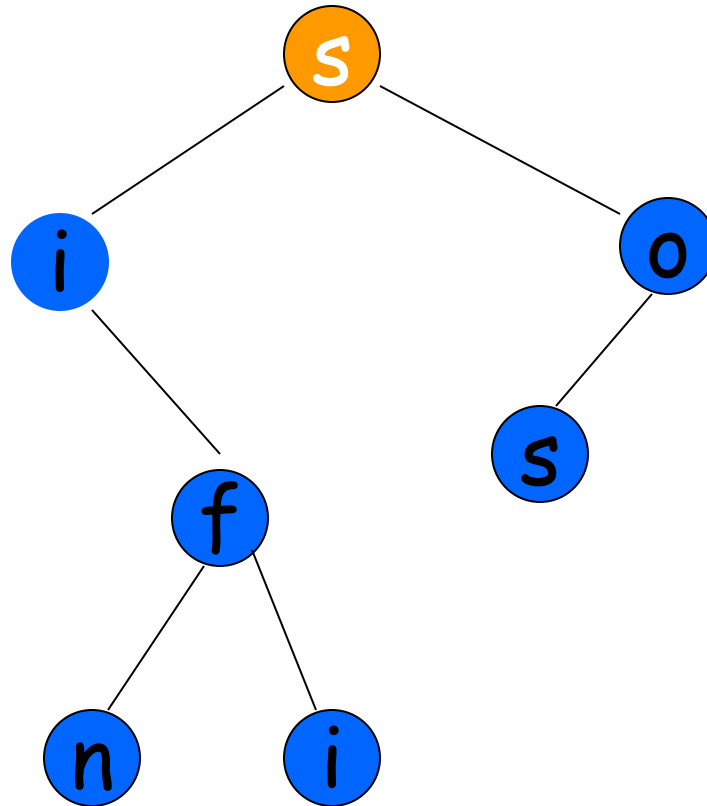
attraversamenti degli alberi = modi di visitare tutti i loro nodi

in profondità = depth-first

ma anche in larghezza = breath-first

percorso **infisso**:

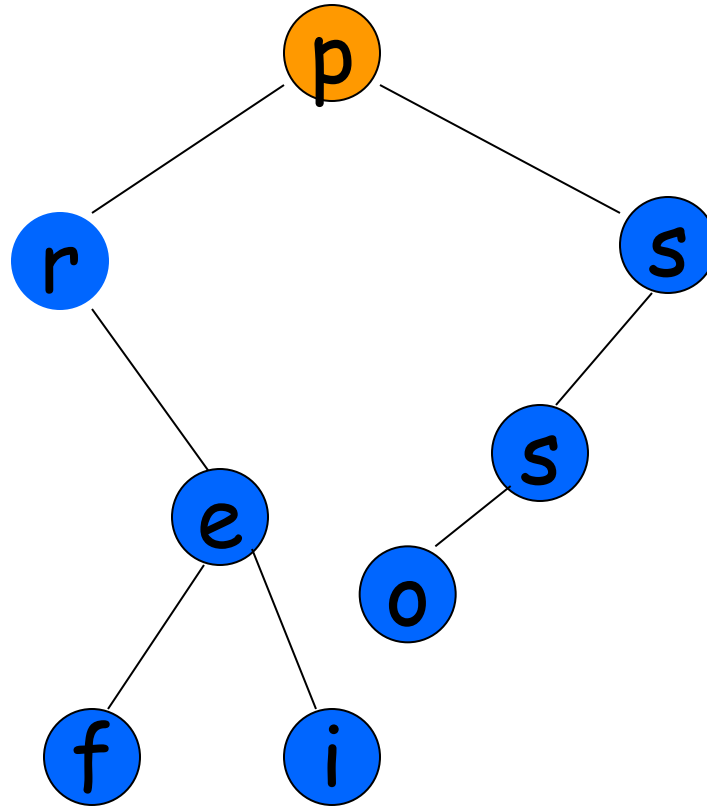
1. a sinistra
2. nodo
3. a destra



in profondità da sinistra a destra

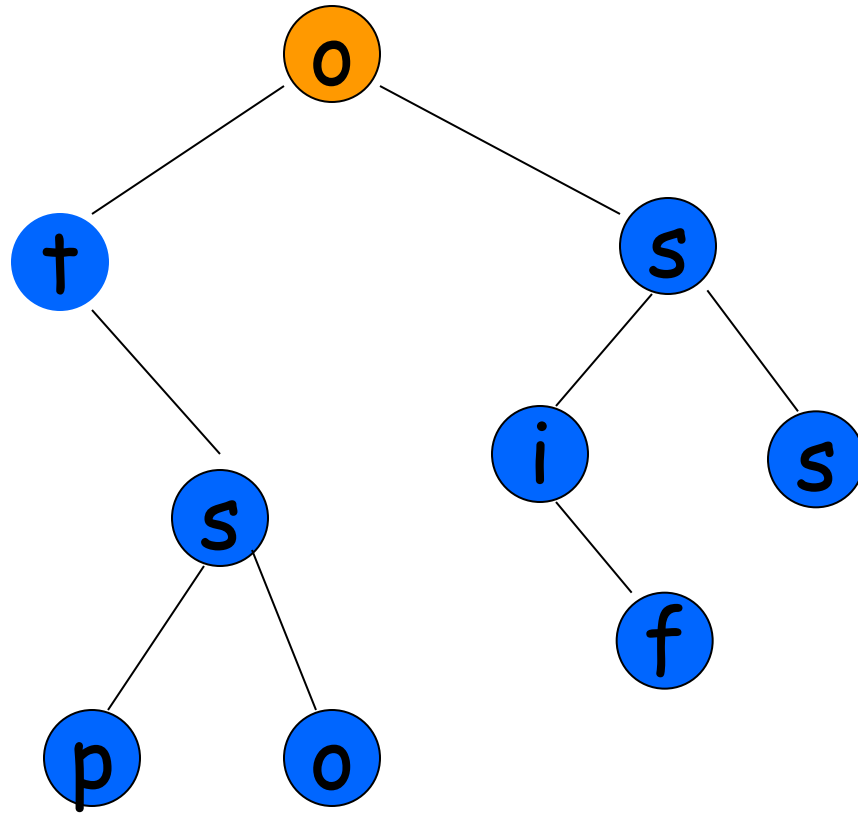
percorso **prefisso**:

1. nodo
2. a sinistra
3. a destra

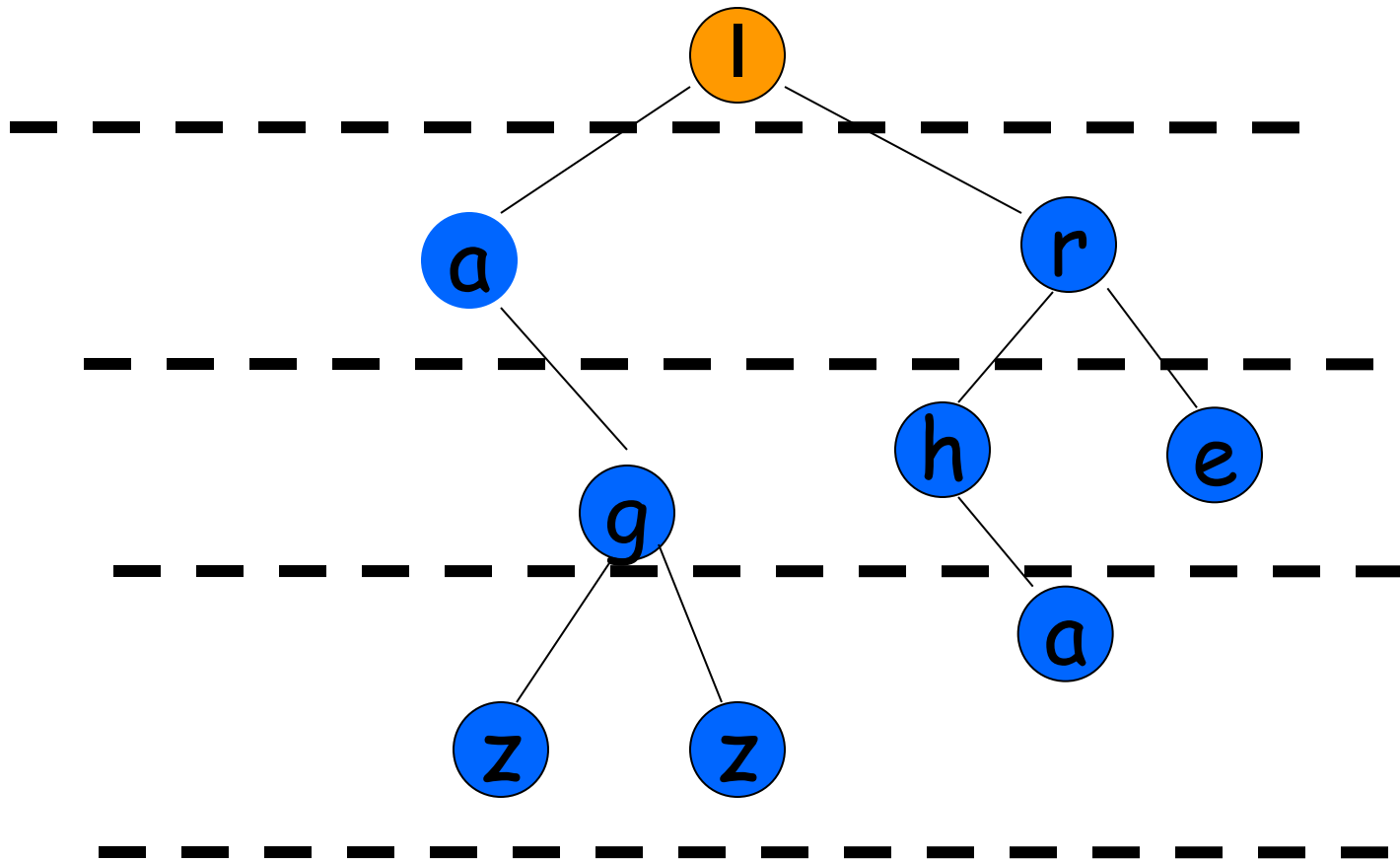


percorso **postfisso**:

1. a sinistra
2. a destra
3. nodo



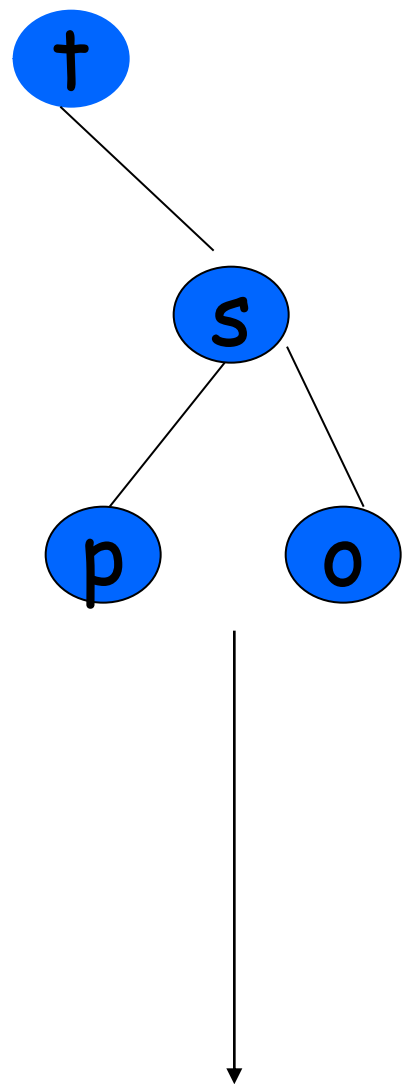
in larghezza



come realizzare un nodo di un albero binario in C++:

```
struct nodo{  
    char info;  
    nodo* left, *right;
```

```
nodo(char a='\0', nodo*b=0, nodo* c=0)  
{info=a; left=b; right=c;}  
};
```



costruiamo questo albero:

```
nodo * root=new nodo('t',0,0);
```

```
root→right=new nodo();
```

```
root→right→info='s';
```

```
root→right→left=new nodo('p',0,0);
```

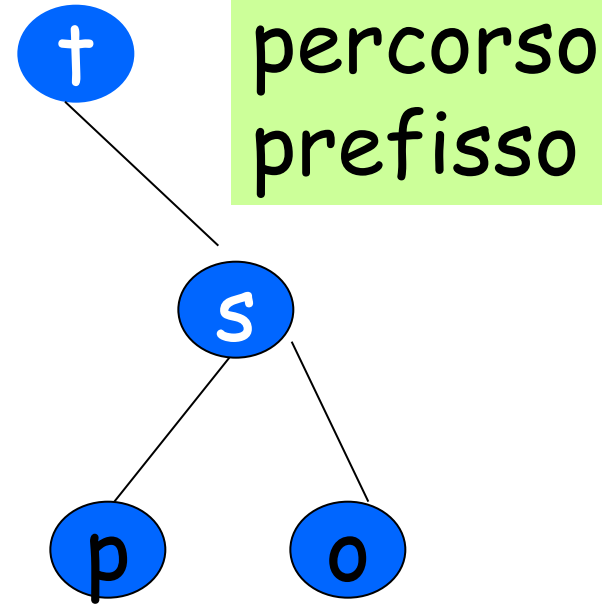
```
root→right→right=new nodo('o',0,0);
```

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

```

void stampa_l(nodo *r)
{
    if(r)
    {
        cout<<r->info<<'(';
        stampa_l(r->left);
        cout<<',';
        stampa_l(r->right);
        cout<<')';
    }
    else
        cout<< '_';
}

```



$t(_, s(p(_, _), o(_, _)))$

stampa in ordine infisso:

```
void infix(nodo *x){  
    if(x) {  
        infix(x->left); // stampa albero sinistro  
        cout<<x->info; // stampa nodo  
        infix(x->right); // stampa albero destro  
    }  
}
```

invocazione: `infix(root);`

trovare e restituire un nodo con un campo
info == y

```
nodo* trova(nodo *x, char y){
```

```
if(!x) return 0;
```

```
if(x->info==y) return x;
```

```
nodo * z= trova(x->left,y);
```

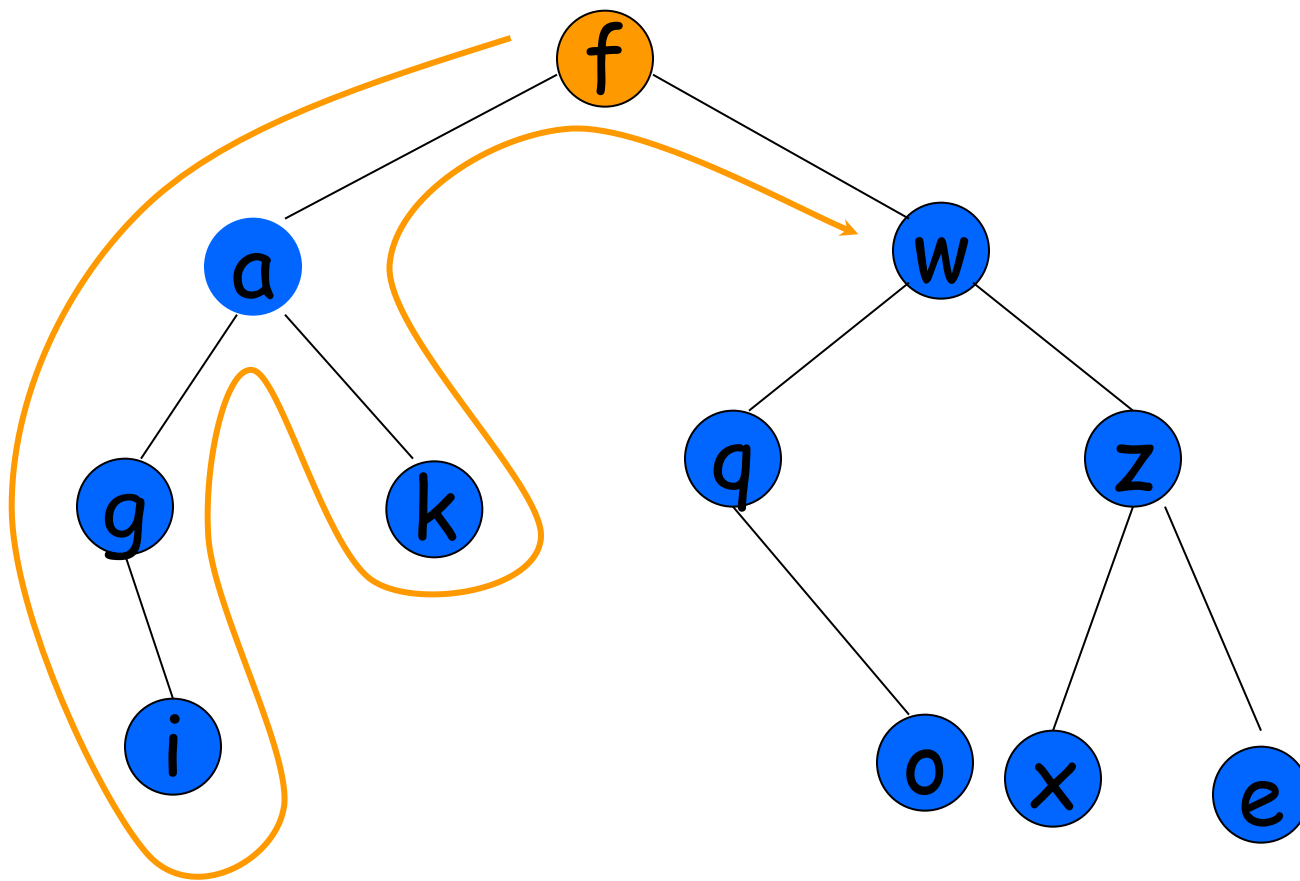
```
if(z) return z;
```

```
return trova(x->right,y);
```

```
}
```

invocazione

```
nodo *w=trova(root,y)
```



cerchiamo 'w'

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

altezza di un albero = profondità massima
dei suoi nodi = distanza massima tra 2
nodi dell'albero

 altezza 0

 altezza 1

albero vuoto? per convenzione -1

PRE=(albero(x) è corretto)

```
int altezza(nodo *x)
```

```
{
```

```
if(!x) return -1;  //albero vuoto
```

```
else {
```

```
    int a=altezza(x->left);
```

```
    int b=altezza(x->right);
```

```
    if(a>b) return a+1;
```

```
    return b+1;
```

```
}}POST=(restituisce l'altezza di albero(x))
```

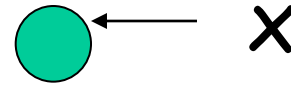
proviamo che è corretto:

base albero vuoto $x = -1$

```
int altezza(nodo *x){  
  if(!x) return -1;  
  else {  
    int a=altezza(x->left);  
    int b=altezza(x->right);  
    if(a>b) return a+1;  
    return b+1;  
  }  
}
```

-1 OK

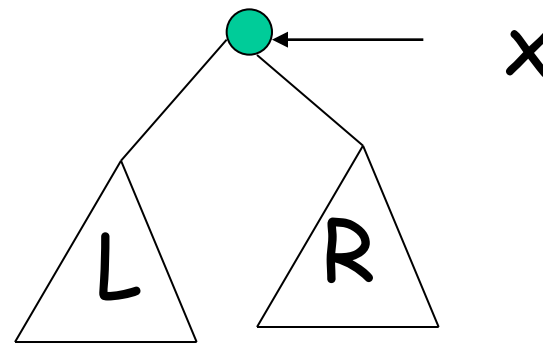
un solo nodo



```
int altezza(nodo *x){  
    if(!x) return -1;  
  
    else {  
        int a=altezza(x->left);    a = -1  
        int b=altezza(x->right);    b = -1  
        if(a>b) return a+1;  
        return b+1;}                return 0  
}
```

OK

in generale:



```
int altezza(nodo *x){  
  if(!x) return -1;
```

```
  else {
```

```
    int a=altezza(x->left);
```

```
    int b=altezza(x->right);
```

```
    if(a>b) return a+1;
```

```
    return b+1;}
```

```
}
```

per ipotesi induttiva

altezza di L

altezza di R

maggiore delle 2
+ 1 OK

potremmo anche evitare di considerare l'albero vuoto per l'altezza.

$PRE = (\text{albero}(x) \text{ corretto e non vuoto})$

PRE=(albero(x) corretto **non vuoto**)

int altezza(nodo* x)

{

if(!x->left && !x->right)

return 0;

int a=-1, b=-1;

if(x->left) a=altezza(x->left);

if(x->right) b=altezza(x->right);

if(a<=b) return b+1;

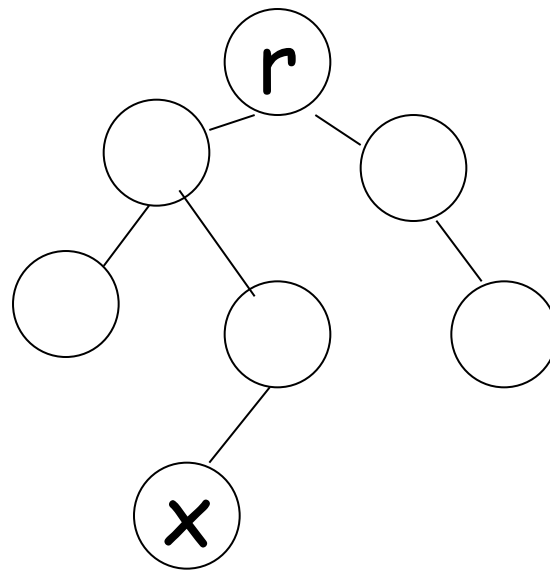
else return a+1;

} POST=(restituisce l'altezza di albero(x))

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

0=sinistra 1= destra

array int $C[]$ e il valore **lung** indica la lunghezza della sequenza:



cammino da r a x:

$C=[010]$ lung=3

cammino da r a r $C=[]$ e lung =0

Problema

dato un array C che contiene una sequenza di 0 e 1 e un albero restituire il nodo corrispondente....se c'è

invocazione: `nodo *z= trova(root, C, lung);`

PRE=(albero(x) corretto, $C[0..lung-1]$ def e 0/1)

nodo * trova(nodo *x, int* C, int lung)

{ if(!x) return 0; // fallito
if(!lung) return x; //trovato

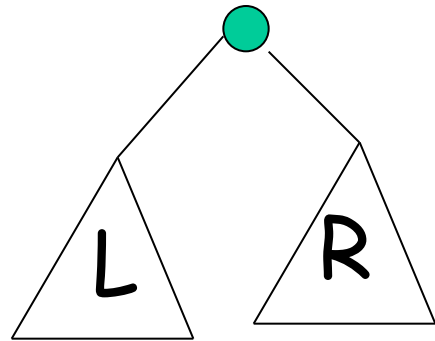
if(*C==0) return trova(x->left, C+1, lung-1);
else
return trova(x->right, C+1, lung-1);
}

POST=(restituisce punt. a nodo alla fine del cammino $C[0..lung-1]$, se c'è in albero(x), e altrimenti 0

inserimento di un nuovo nodo in un
albero: il nuovo nodo va inserito come
figlio di un nodo già esistente e diventa
quindi una foglia

o l'unico nodo se l'albero era vuoto

inseriamo sempre nel sotto albero che
contiene meno nodi



cioè conto i nodi
di L e di R ed
inserisco il nodo
nel + piccolo dei
due

```
int conta_n(nodo*r)
{
    if(!r) return 0;
    else
        return conta_n(r->left)+conta_n(r->right)+1;
}
```

PRE=(albero(r0) corretto e y def)

nodo* insert(nodo*r0, int y)

POST= sia $rn = \text{nodo}(y, 0, 0)$, insert restituisce $\text{albero}(r0) + rn$ t.c. se il cammino dalla radice $r0$ a rn è $[r0, r1, r2, \dots, rn]$

- 1) per ogni i in $1..n$, ri è il figlio di $r(i-1)$ nel cui albero ci sono meno nodi,
- 2) in $\text{albero}(r0)$ non c'era nodo al posto di rn

e se $r0=0$???

```
nodo* insert(nodo*r, int y)
{
    if(!r) return new nodo(y);

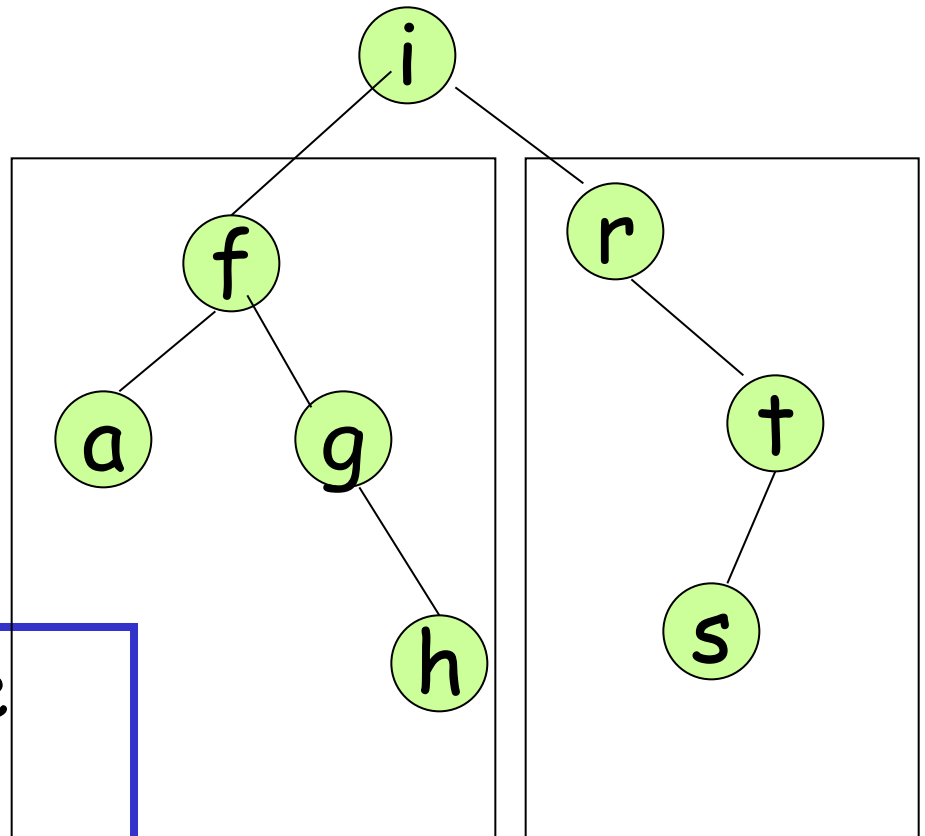
    if(conta_n(r->left)<=conta_n(r->right))
        r->left=insert(r->left,y);
    else
        r->right=insert(r->right,y);
    return r;
}
```

osservare similarità con l'inserimento in fondo
ad una lista concatenata: **operazioni inutili**

passaggio per riferimento

```
void insert(nodo*& r, int y)
{
    if(!r) r= new nodo(y);
    else
    if(conta_n(r->left)<=conta_n(r->right))
        insert(r->left,y);
    else
        insert(r->right,y);
}
```

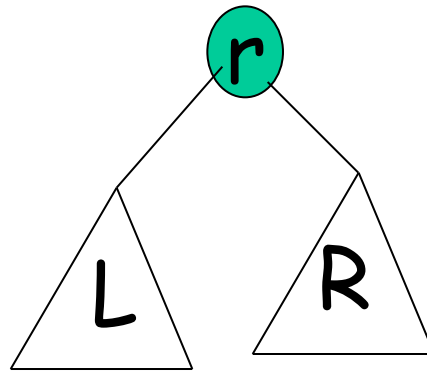
binary **search** trees (BST):



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

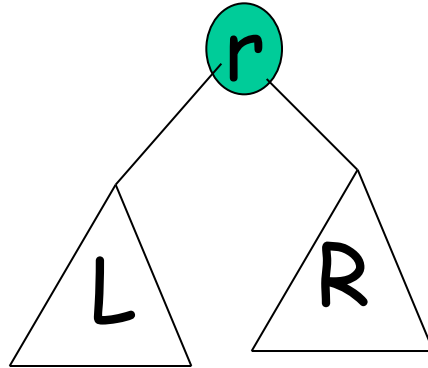
in un BST è **facile** (efficiente) trovare
un nodo con un certo campo info y
e restituire il puntatore a quel nodo se
lo troviamo
e 0 altrimenti

non BST:



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 più semplice:



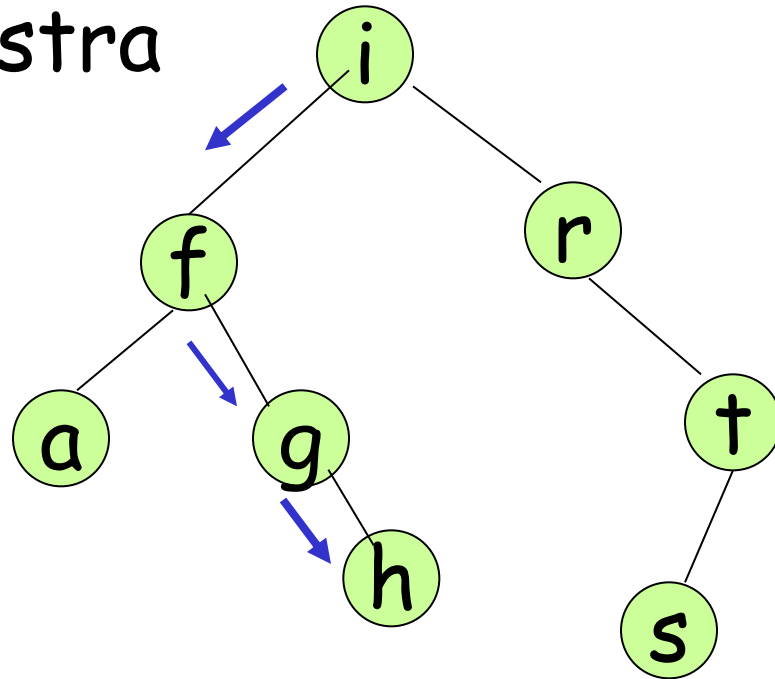
1. $r \rightarrow \text{info} == y$ restituisco r , altrimenti:
2. se $r \rightarrow \text{info} > y \rightarrow$ cerco solo in L
altrimenti cerco solo in R

cerchiamo h:

$h < i$ andiamo a sinistra

$h > f$ destra

$h > g$ destra



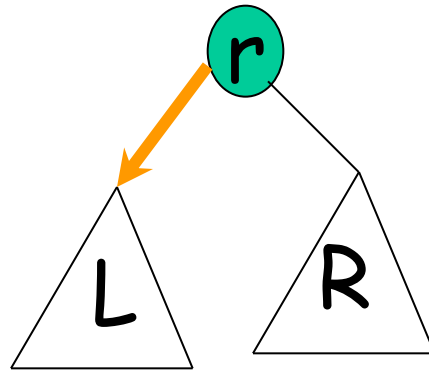
trovato !!!

ricerca in un BST:

```
nodo *search(nodo *x,char y){  
    if(!x) return 0;  
    if(x->info==y) return x;  
    if(x->info>y)  
        return search(x->left,y);  
    else  
        return search(x->right,y);  
}
```

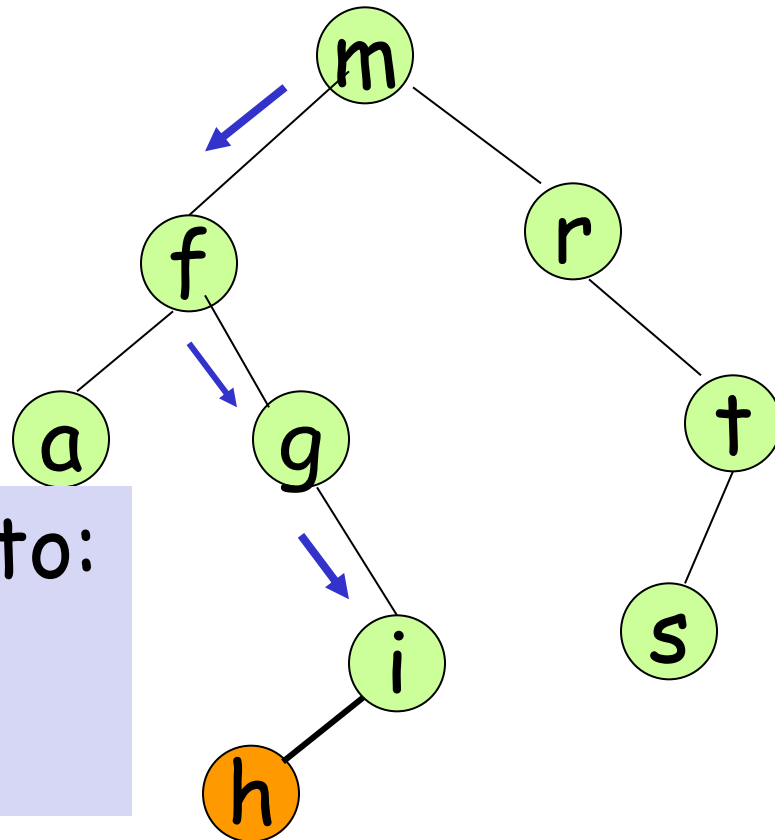
quante chiamate ricorsive si fanno al massimo?

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?



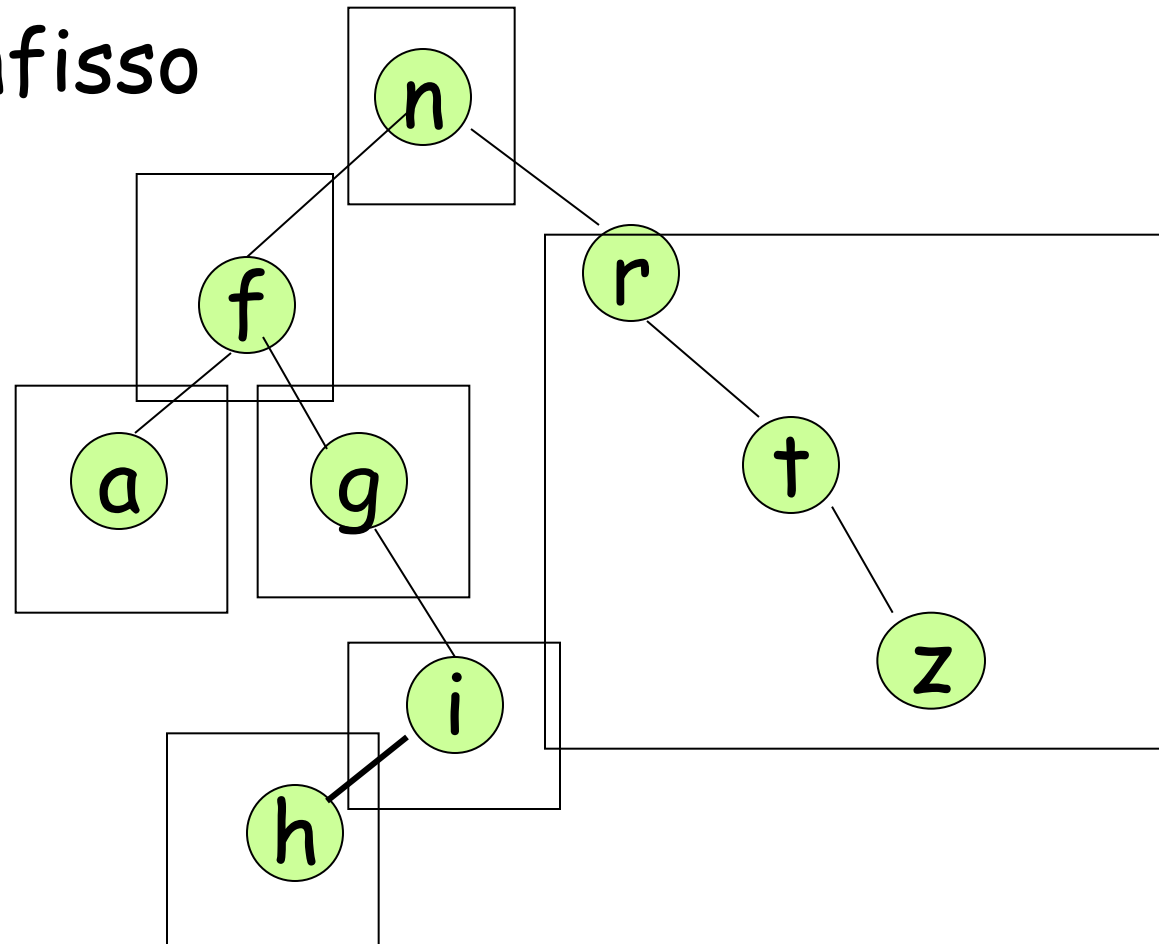
idea per l'inserimento:
lo inseriamo dove lo
cerchiamo

inserimento in un BST:

```
nodo * insert(nodo *r, int y){  
  if(!r) return new nodo(y,0,0);  
  if(r→info > y)  
    r→left=insert(r→left, y);  
  else  
    r→right=insert(r→right, y);  
  return r;  
}
```

realizzate la soluzione col passaggio di r per riferimento

percorso infisso



a f g h i n r t z

sono in ordine !!