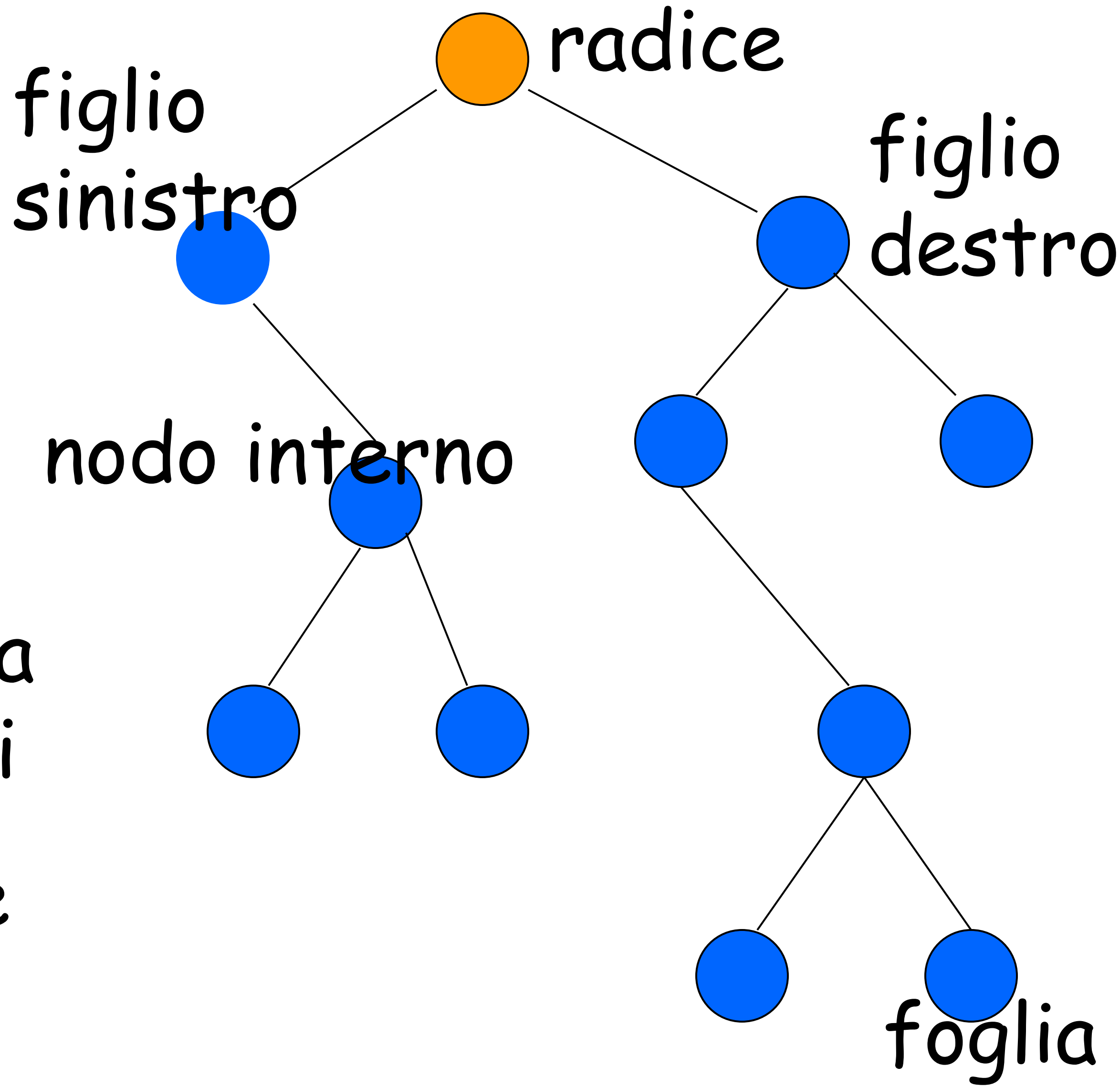
alberi binari e ricorsione

cap. 12

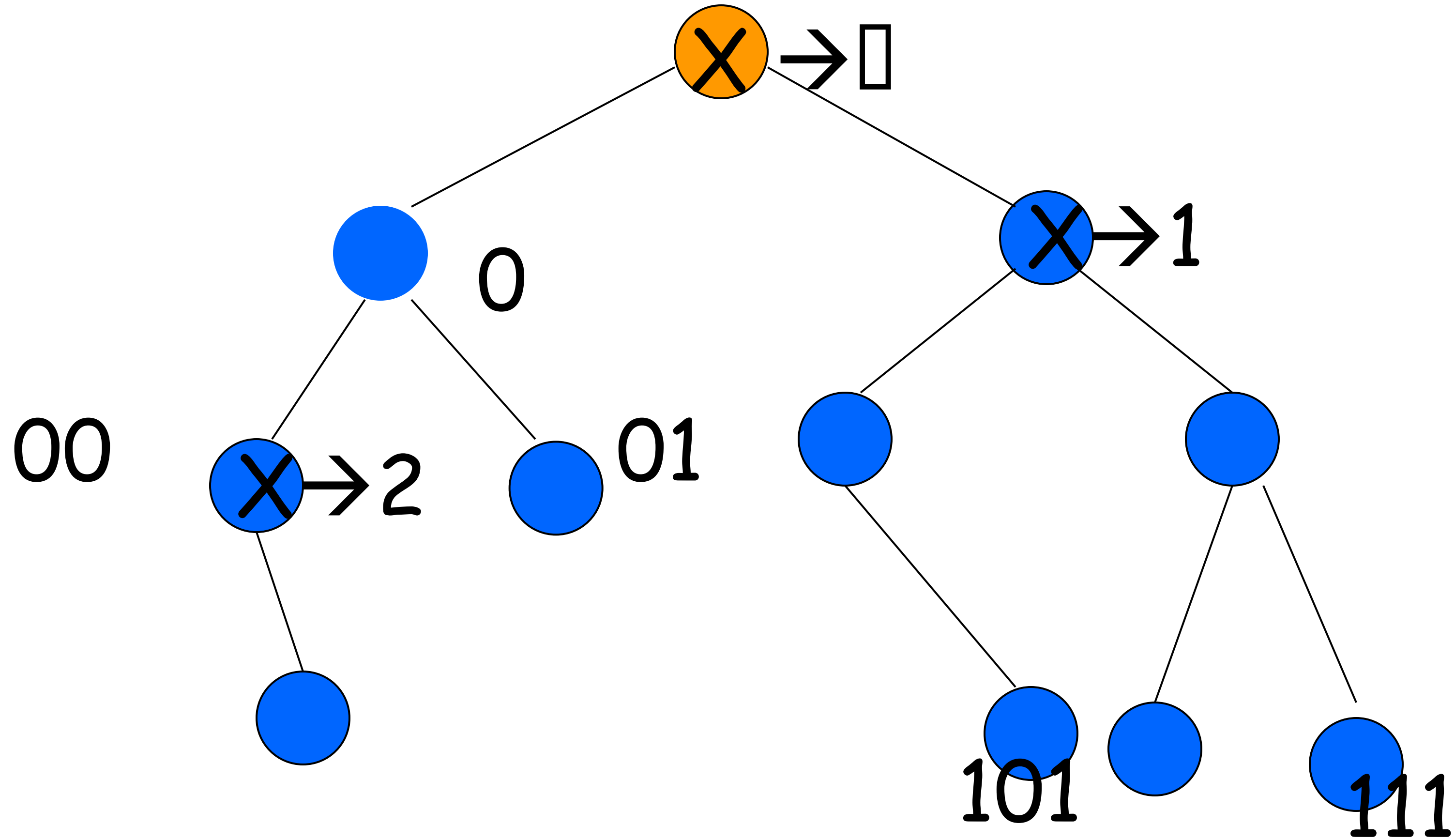
un albero binario:



ogni nodo ha al più 2 figli

ogni figlio è destro o sinistro

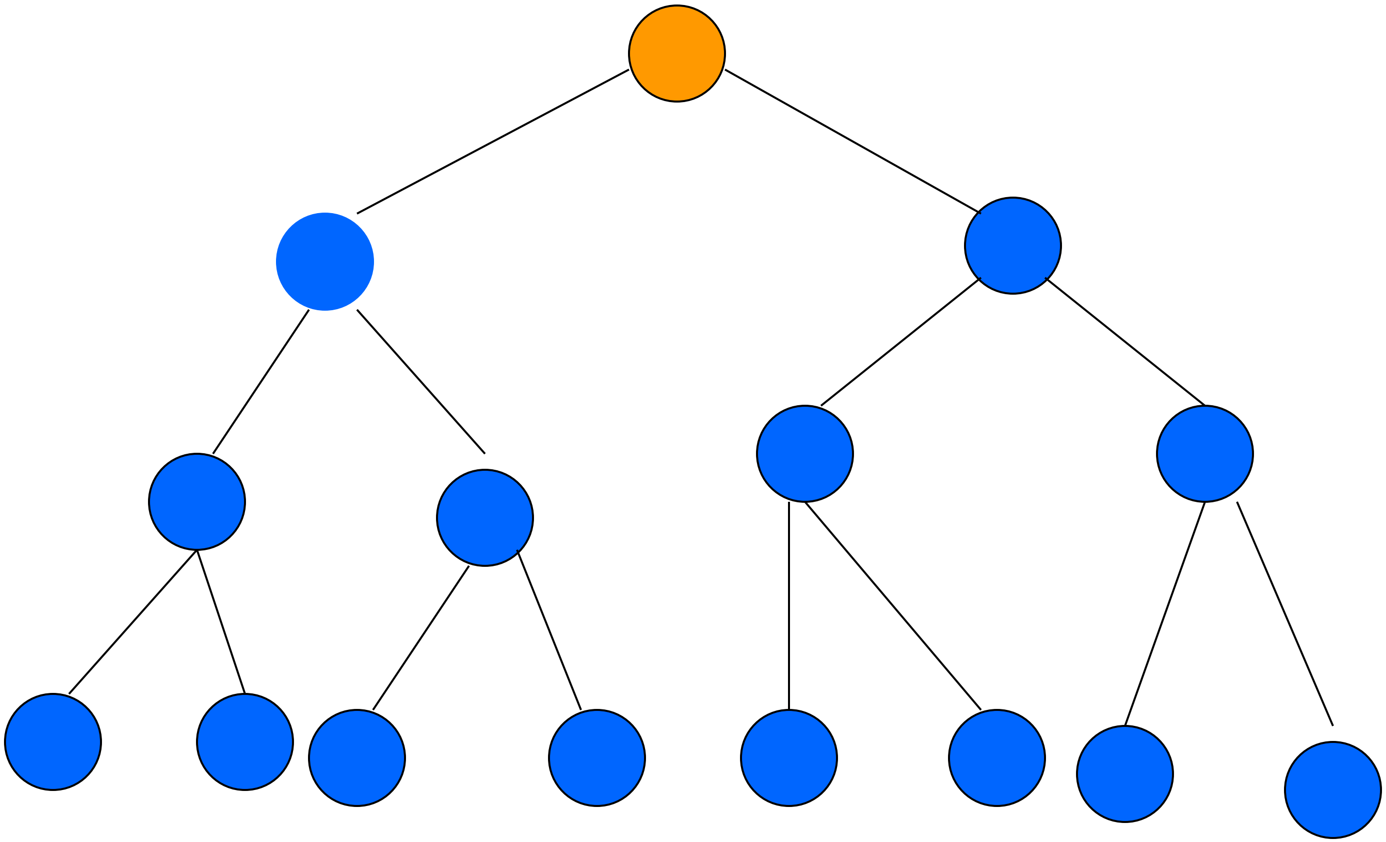
cammini = sequenze di nodi = sequenze di 0 e 1



profondità di un nodo

altezza dell'albero=prof. max delle foglie

albero binario completo



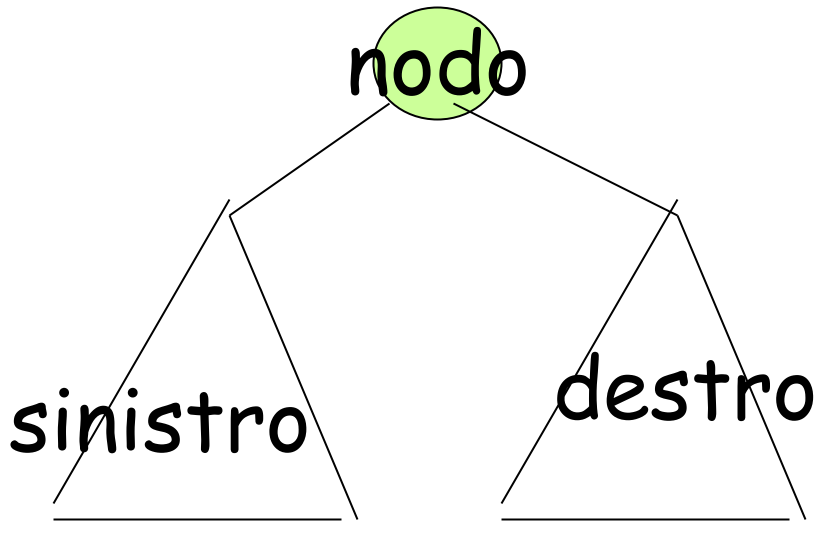
ogni livello è completo, se h= altezza

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

definizione ricorsiva degli alberi:

albero binario è:

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



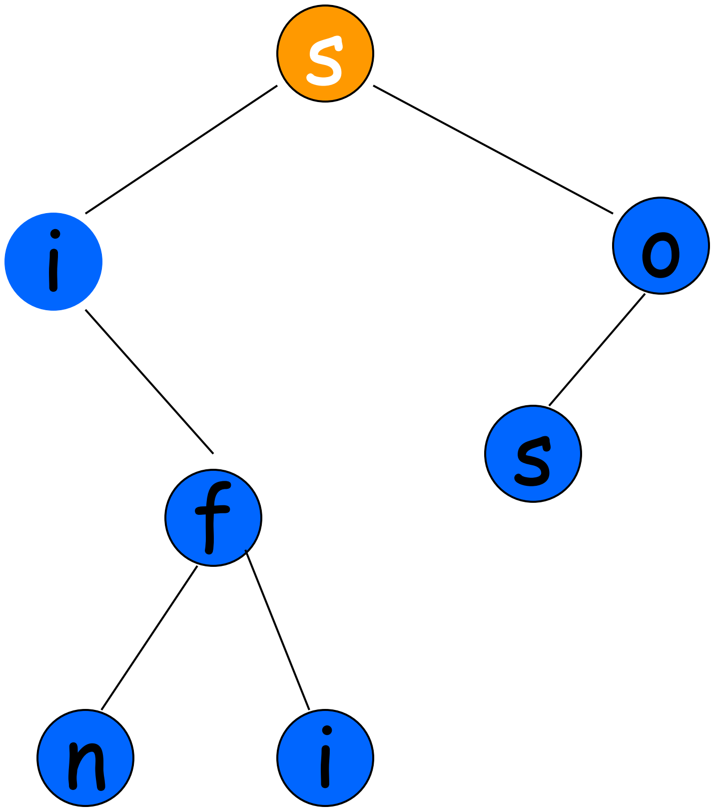
attraversamenti degli alberi = modi di visitare tutti i loro nodi

in profondità = depth-first

ma anche in larghezza = breath-first

percorso infisso:

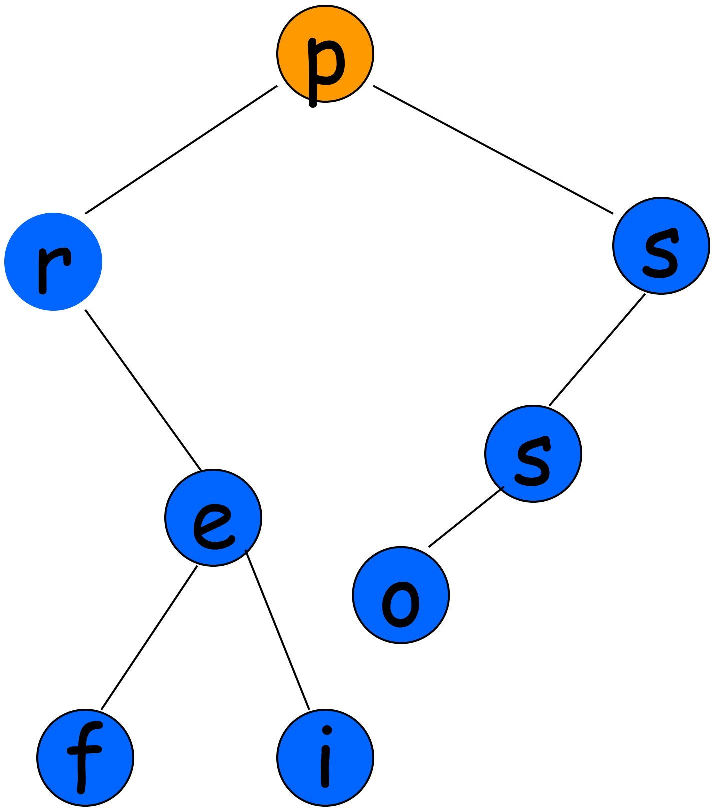
1. a sinistra
2. nodo
3. a destra



in profondità da sinistra a destra

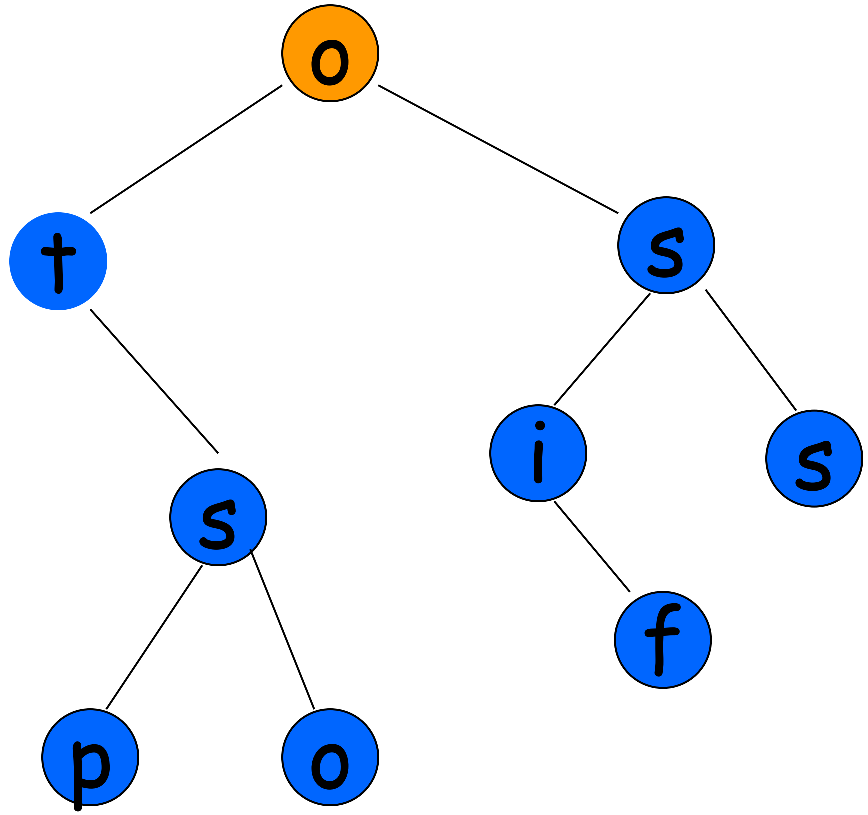
percorso prefisso:

1. nodo
2. a sinistra
3. a destra

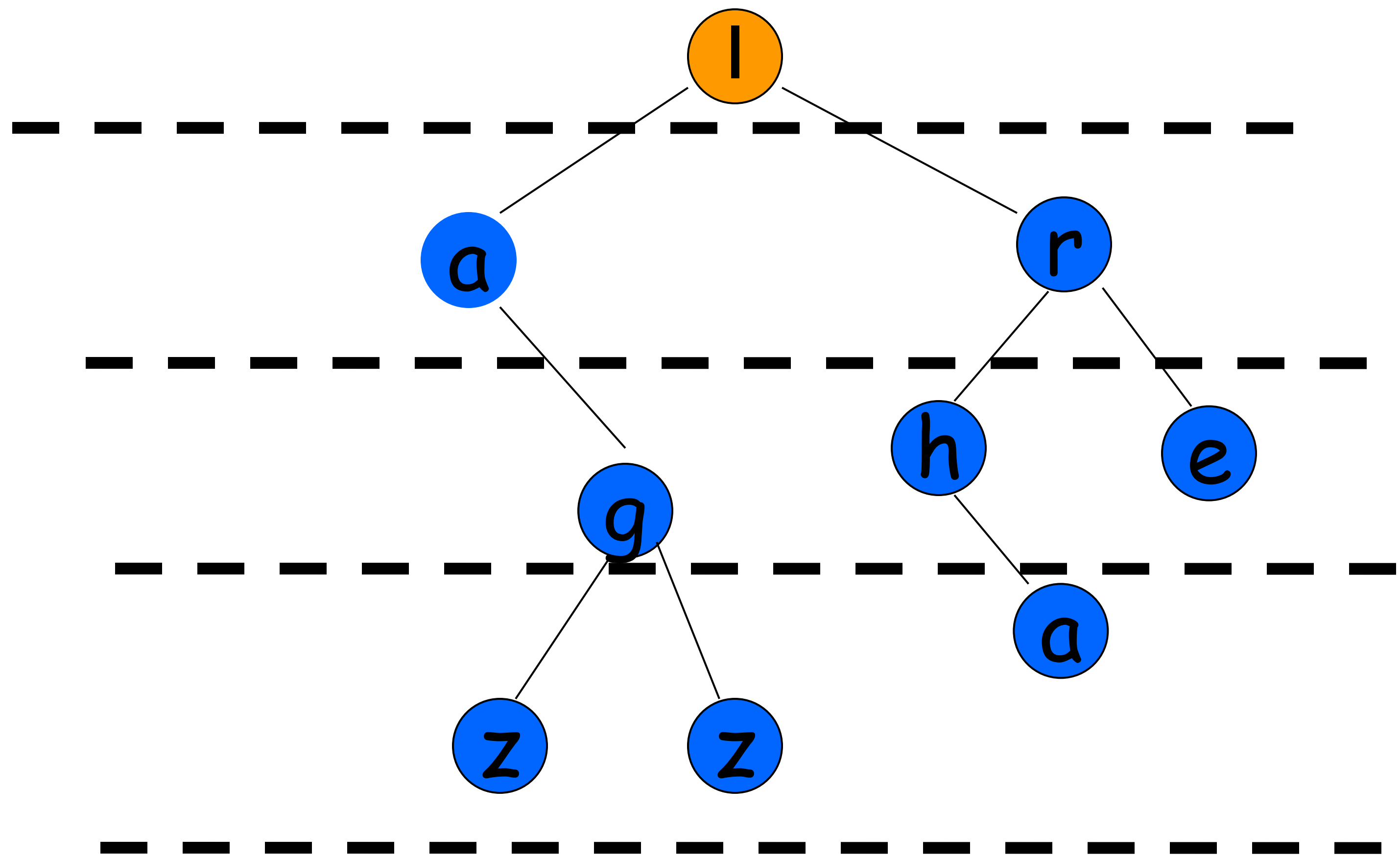


percorso postfisso:

1. a sinistra
2. a destra
3. nodo

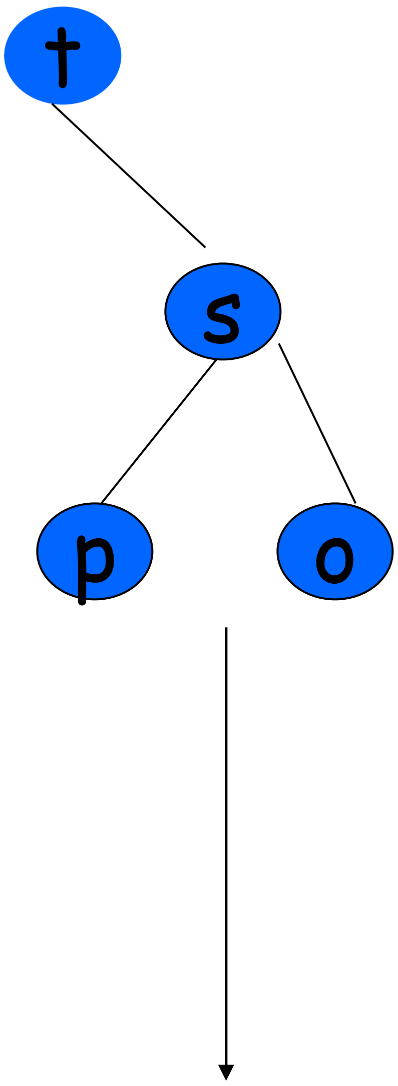


in larghezza



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

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

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

};
```



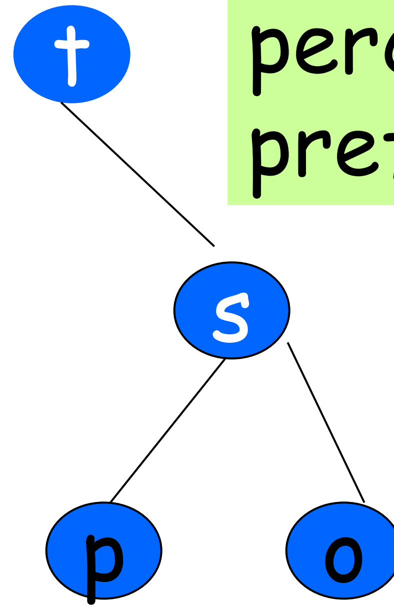
costruiamo questo albero:

```
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<< '_';
}
```

percorso prefisso



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

stampa in ordine infisso:

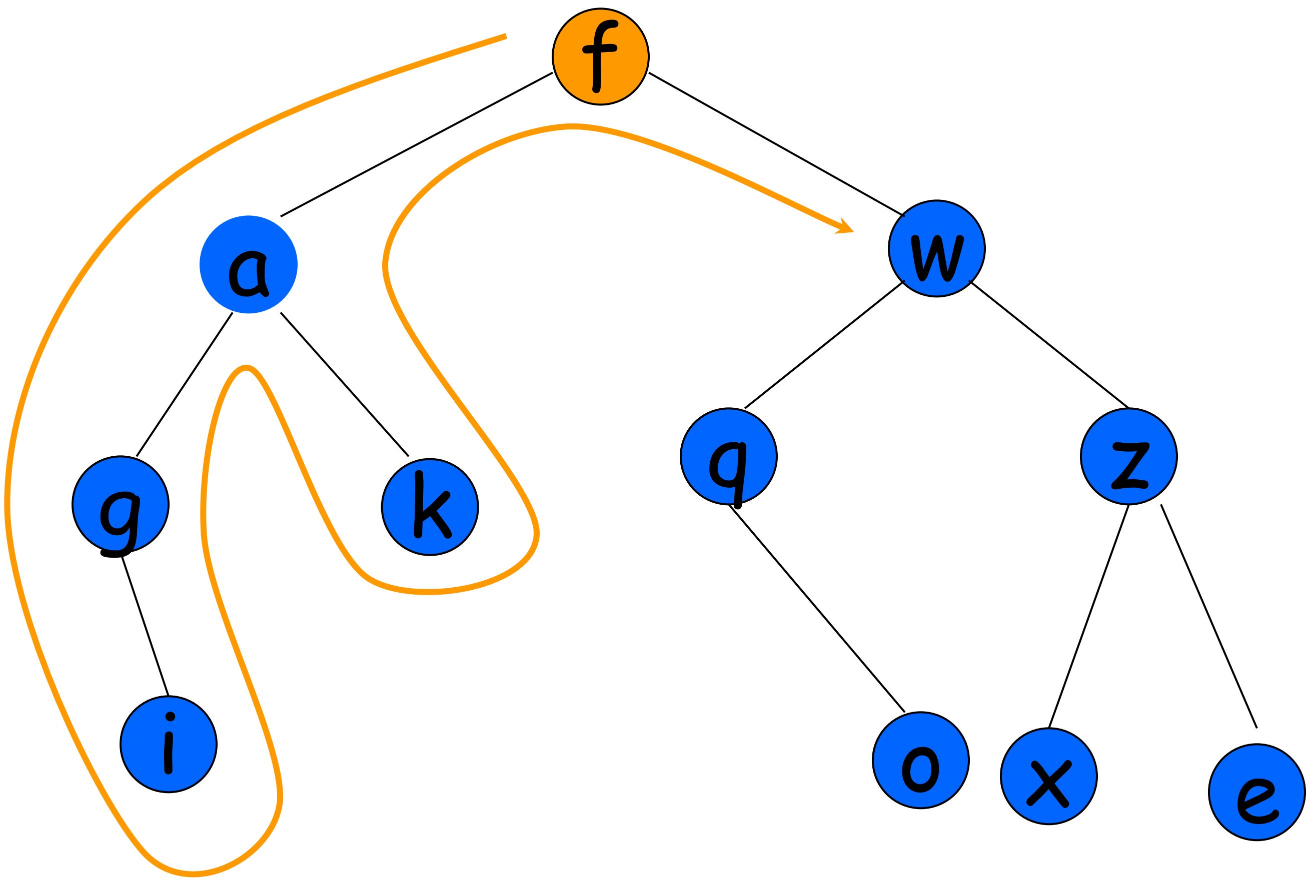
```
void infix(nodo *x){
if(x) {
infix(x->left);   // stampa albero sinistro
cout<<x->info; // stampa nodo
infix(x->right);  // stampa albero destro
}}

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);
}
```

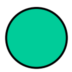
invocazione

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

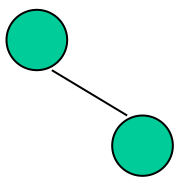


cerchiamo 'w'

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

calcolo dell'altezza:

```
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;
}}
```

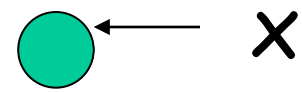
proviamo che è corretto:

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

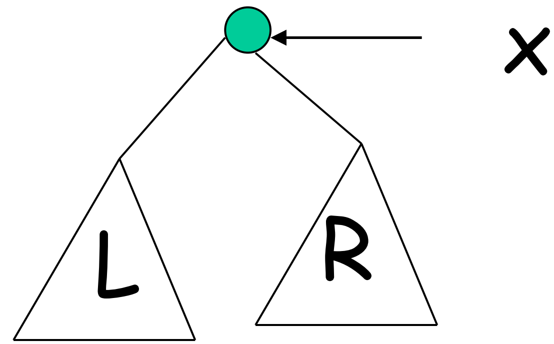
un solo nodo




```
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

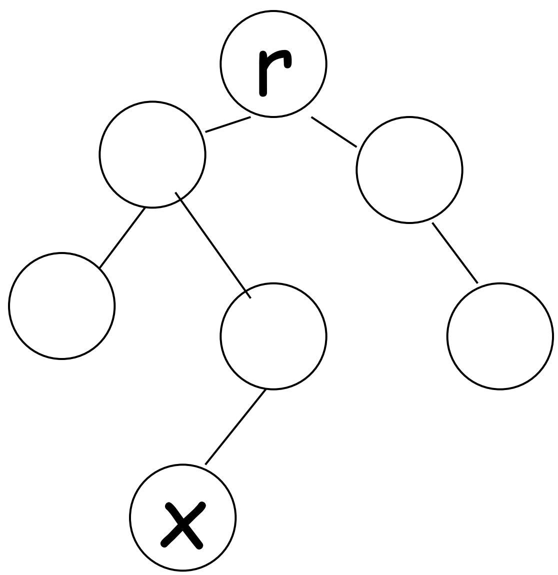
+ 1 OK

```
PRE=(albero corretto non vuoto)
int altezza(nodo* r)
{
  if(!r->left && !r->right)
      return 0;
  int a=-1, b=-1;
  if(r->left) a=altezza(r->left);
  if(r->right) a=altezza(r->right);
  if(a<=b) return b+1;
    else return a+1;
} POST=(restituisce l'altezza di r)
```

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 per x:

C=[010] lung=3

cammino di r C=[] e lung =0

dato un array C che contiene un cammino, restituire il nodo corrispondente

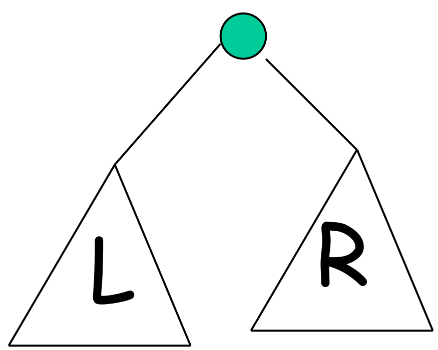
```
nodo * trova(nodo *x, int*  C, int lung)
{ if(!x) return 0;  // fallito
if(!lung) return x;  //trovato
if(*C==0)
return trova(x->left, C+1, lung-1);
else
return trova(x->right,C+1, lung-1);
}
```

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

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;


}
```

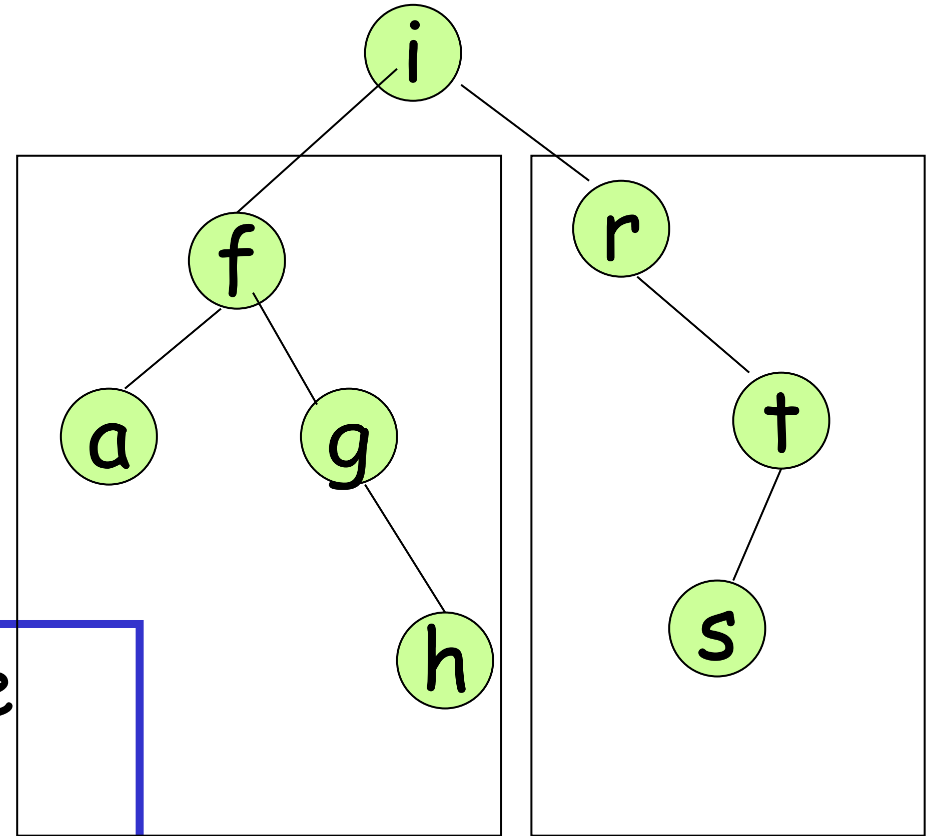
```
nodo* insert(nodo*r, int y)
{
  if(!r) return new nodo(y);

  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 inserimento in fondo a lista

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):

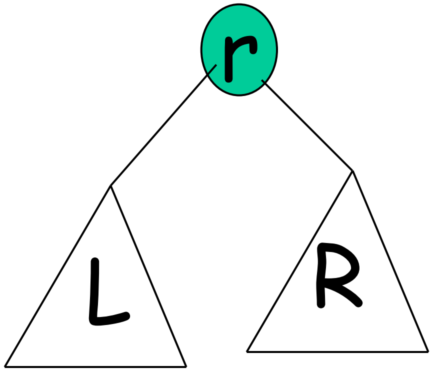


ogni nodo è maggiore dei nodi nel suo sottoalbero sinistro e minore di quelli del suo sottoalbero destro in un BST è facile (efficiente) trovare un nodo con un certo campo info y

e restituire il puntatore a quel nodo se lo troviamo

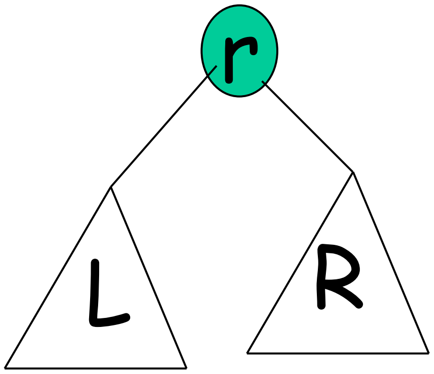
e 0 altrimenti

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→info == y restituisco r, altrimenti:

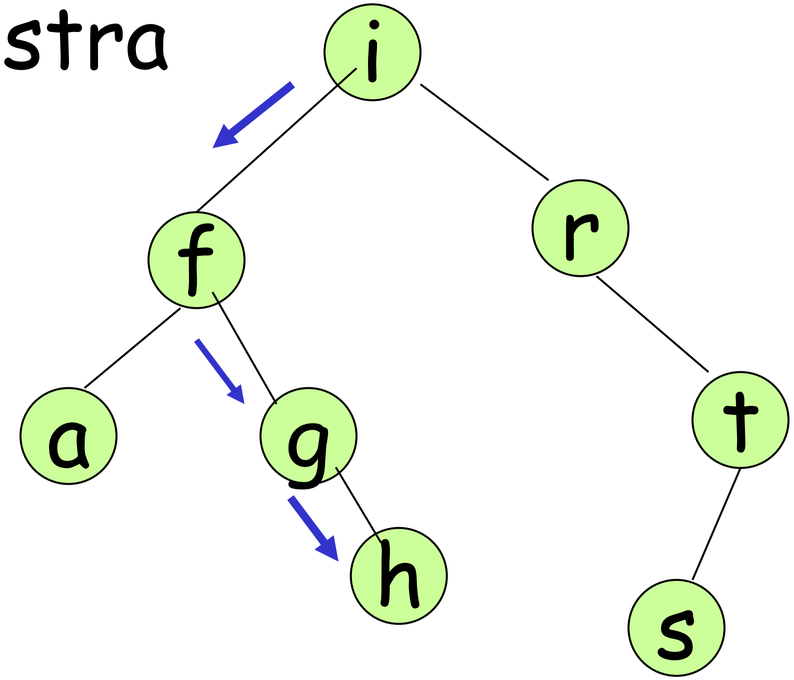
2. se r→info > y ➔ cerco solo in L
   altrimenti cerco solo in R

cerchiamo h:

h<i andiamo a sinistra

h>f destra

h>g destra



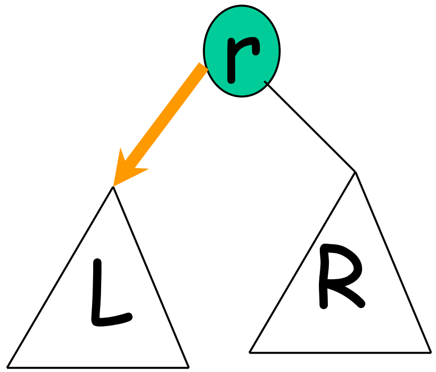
trovato !!!

ricerca in un BST:

```
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

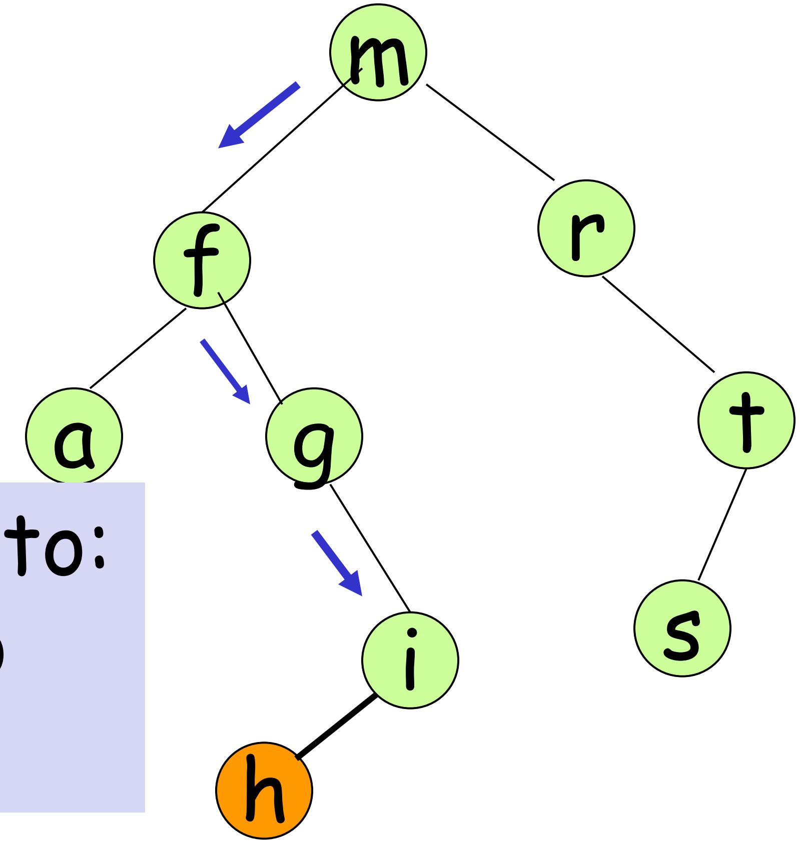


se l'albero è equilibrato, altezza = log n

dove n è il numero dei nodi dell'albero

una bella differenza tra n e log n !!!

se h non ci fosse?

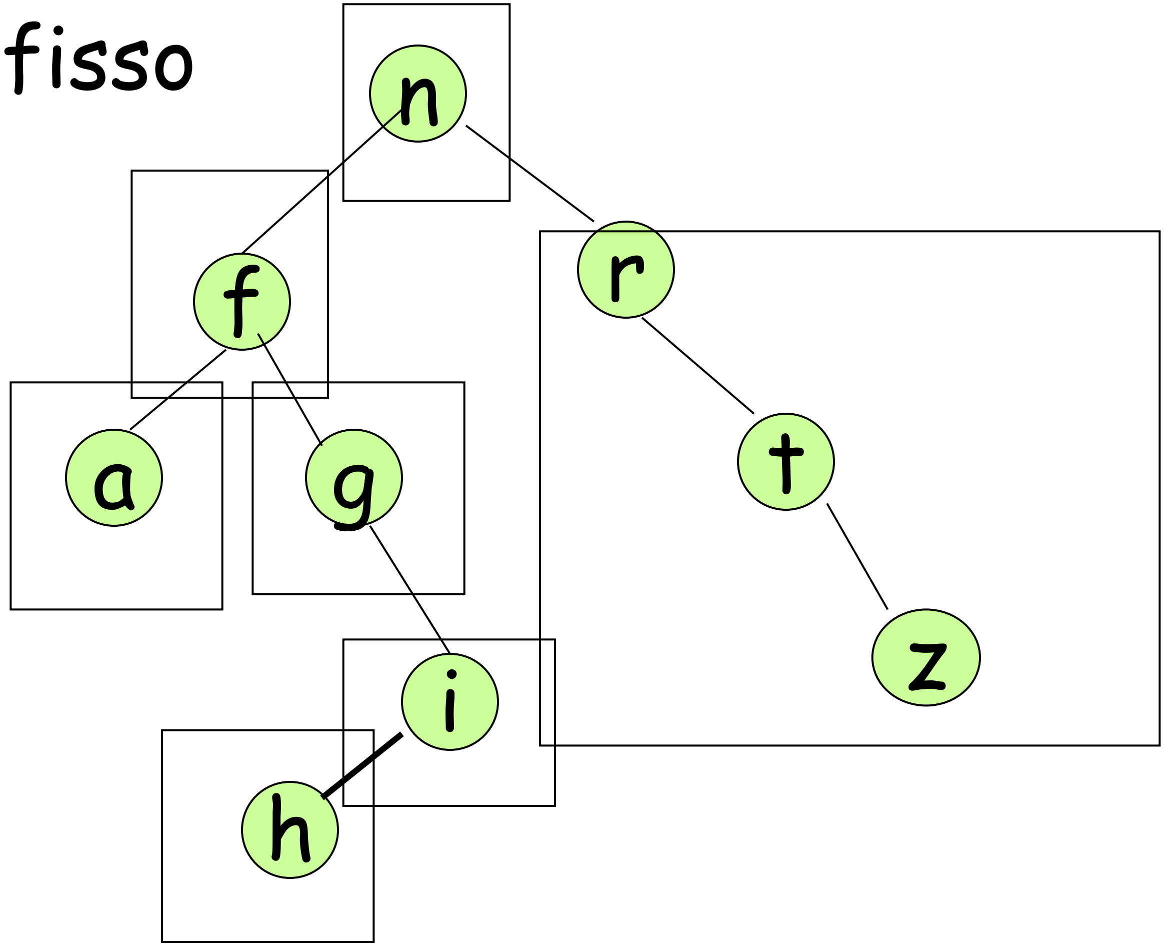


idea per l'inserimento:
lo inseriamo dove lo
cerchiamo

inserimento in un BST:

```
nodo * insert(nodo *r, int y){
if(!r) return new nodo(y,0);
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 !!

inseriamo

10 2 16 20 4 6 8 22 1

partendo dall'albero vuoto:

esercizi sugli alberi binari

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

contare i nodi con esattamente un figlio

```
int cncuf(nodo *x)
{ if(x)
if(!x->left && x->right || x->left &&
        !x->right)
return 1+ cncuf(x->left)+cncuf(x->right);
else
return cncuf(x->left)+cncuf(x->right);
else
return 0;
}
```

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 seguo?

è arbitario purchè si sia in grado di percorrerli tutti

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

```
nodo * prof_data(nodo * r, int k)
{
if(! r ) return 0;

if(k==0) return r;

nodo * p=prof_data(r→left,k-1);
if(p) return p;

return prof_data(r→right,k-1);
}
```

trovare nodo minimo a profondità k

PRE=(r punta ad albero poss. vuoto, k >=0)

nodo* prof(nodo*r, int k)

POST=(restituisce un nodo di r a prof k che campo info minimo tra i nodi a prof. k e se non ci sono nodi a prof. k, restituisce 0)

```
//minimo nodo a prof k
nodo* prof(nodo*r, int k)
{
  if(r)
   {
    if(k==0)
     return r;
    nodo*a=prof(r->left,k-1);
    nodo*b=prof(r->right,k-1);
    if(a && b)
      if(a->info <= b->info)
        return a;
       else
        return b;
```

```
  if(a)
        return a;
      else
        return b;
    }
   else //vuoto
    return 0;
}
```

trovare profondità minima tra le foglie

e poi vogliamo anche una foglia a profondità minima

usiamo:

```
bool leaf(nodo *n)
{return (!n->left && !n->right);}
```

//PRE=(x albero corretto non vuoto, prof def.)

```
int  prof_min(nodo*x, int prof)
```

//POST=(restituisce k t.c. k-prof è profondità minima di una foglia in x)

ATTENZIONE: PRE richiede di fermarci prima di esaurire l'albero !!

```
int  prof_min(nodo*x, int prof)
{if(leaf(x)) return prof;
 int a=INT_MAX,b=INT_MAX;
 if(x->left)
   a=prof_min(x->left,prof+1);
 if(x->right)
   b=prof_min(x->right,prof+1);
   if(a <= b)
     return a;
   else
     return b;
}
```

vogliamo anche il puntatore al nodo:

la funzione restituisce un valore:

struct foglia{nodo* fo; int prof;};

PRE=(x punta ad albero corretto anche vuoto, prof è definito)

non dobbiamo preoccuparci di esaurire l'albero

```
foglia  prof_min(nodo*x, int prof)
{if(x )
        if(leaf(x))
                return foglia(x,prof);
        else
        {foglia a = prof_min(x->left,prof+1);
        foglia b=prof_min(x->right,prof+1);
        if(a.prof>b.prof) return b;
        else
          return a;
        }
   return foglia(0,INT_MAX);
}
```

NOTARE:
niente
allocazione
dinamica
PROBLEMI?

altra soluzione più efficiente:
inutile cercare a profondità k se abbiamo già trovato una foglia a profondità minore o uguale di k

passaggio per riferimento

```
void f(nodo*x,int prof, foglia & m)
{ if(prof>=m.prof) return;
if(x)
        {
          if(leaf(x))
                {m.fo=x; m.prof=prof; return;}
           else
                {
                        f(x->left,prof+1,m);
                        f(x->right,prof+1,m);
                }

        }
}
```

invocazione:

```
foglia p(0,INT_MAX);
f(root,0,p);
```