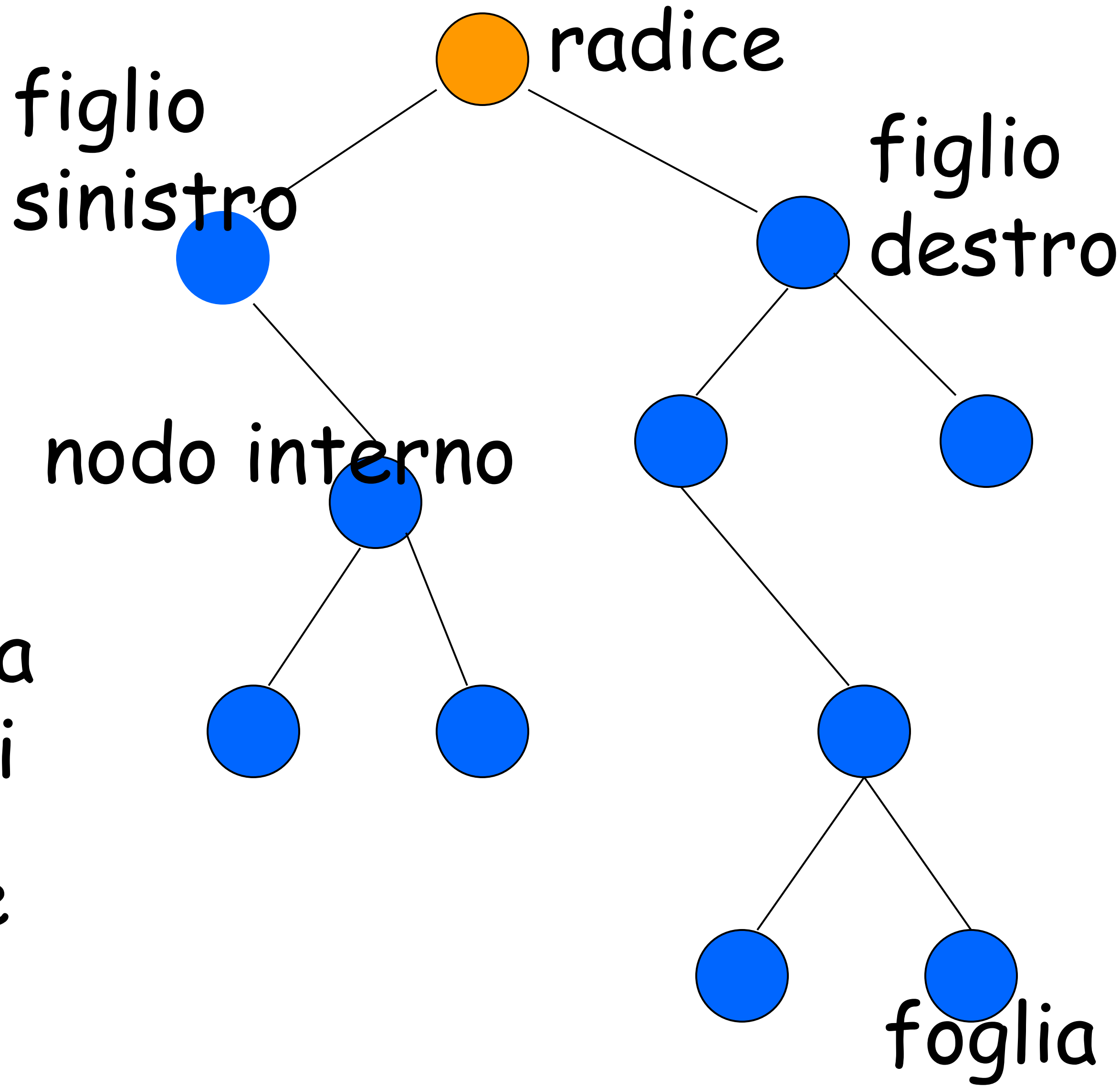
alberi binari e ricorsione

cap. 12

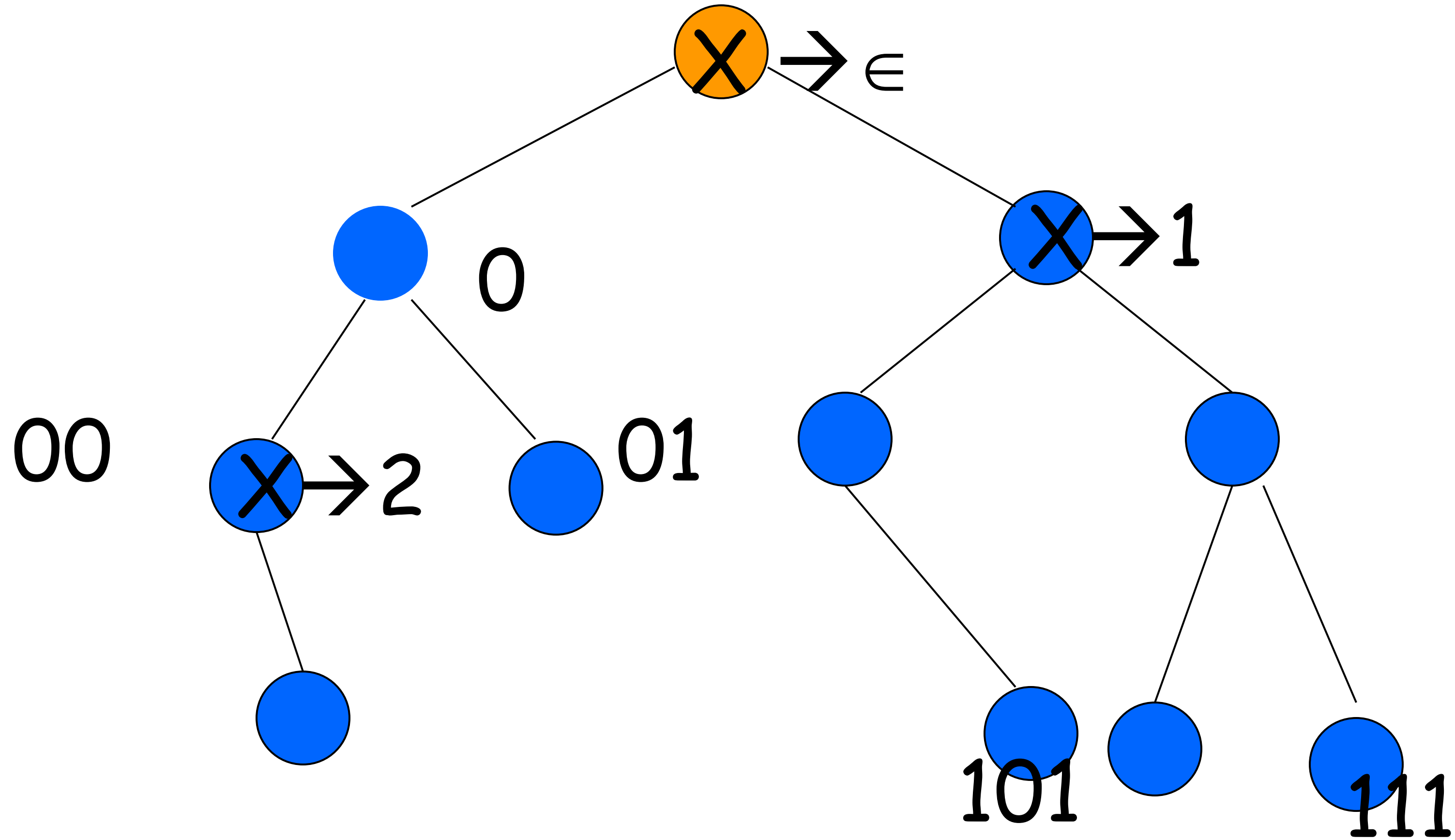
un albero binario:



ogni nodo ha al più 2 figli

ogni figlio è destro o sinistro

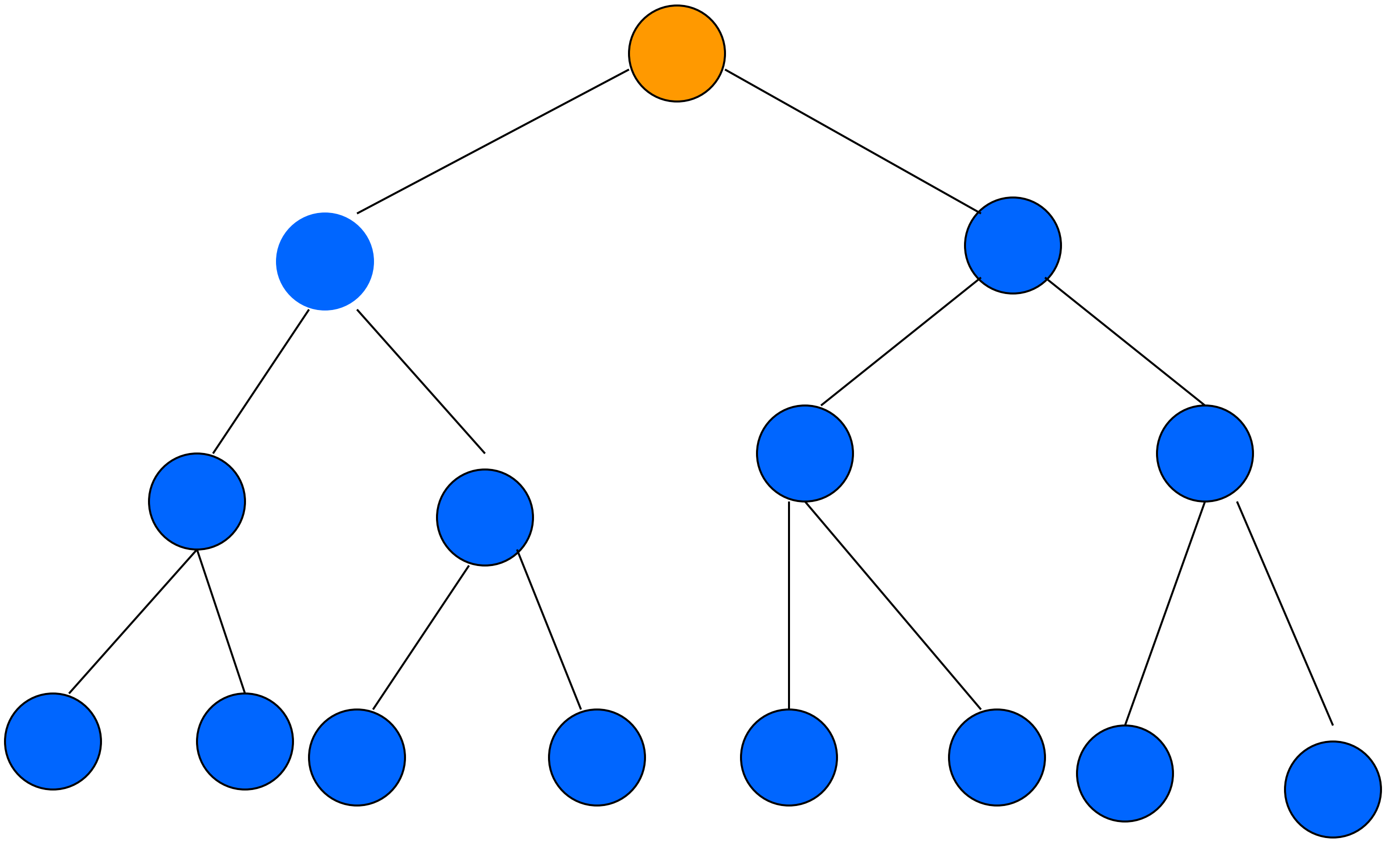
cammini = sequenze di nodi = sequenze di 0 e 1



profondità di un nodo

altezza dell'albero=prof. max delle foglie

albero binario completo



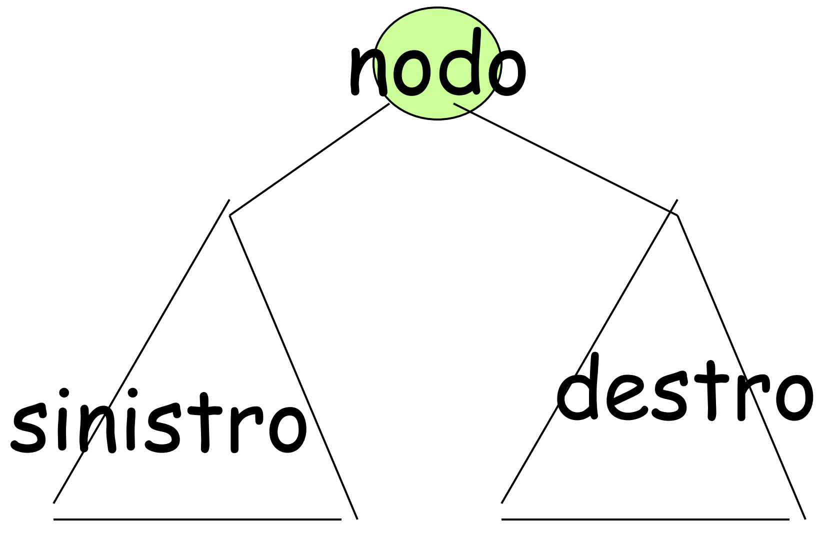
ogni livello è completo, se h= altezza

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

definizione ricorsiva degli alberi:

albero binario è:

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



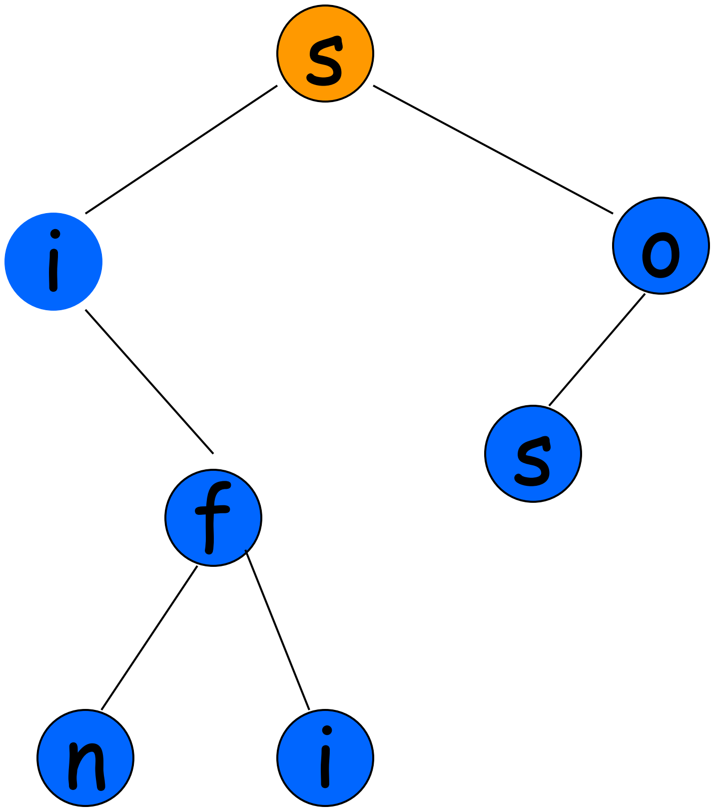
attraversamenti degli alberi = modi di visitare tutti i loro nodi

in profondità = depth-first

ma anche in larghezza = breath-first

percorso infisso:

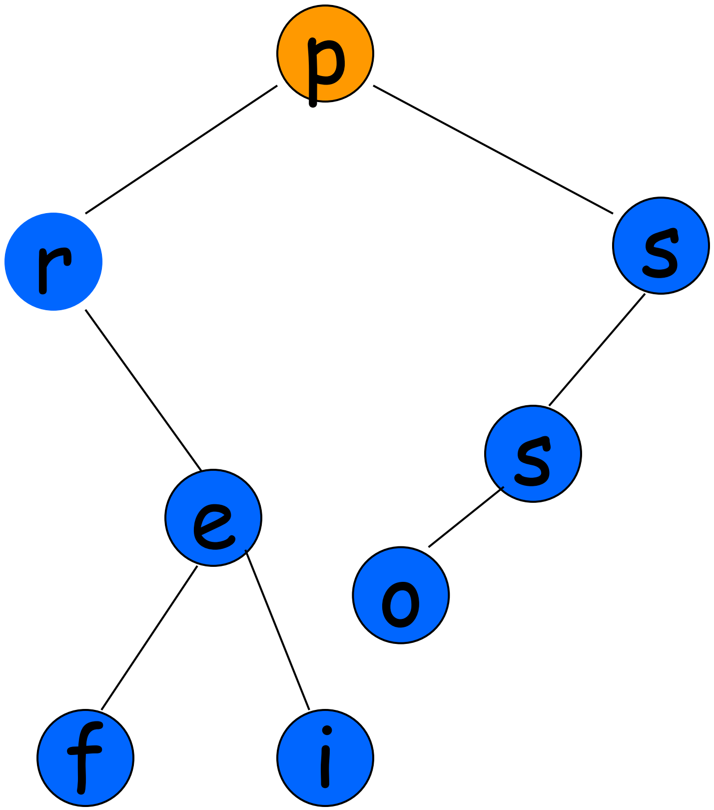
1. a sinistra
2. nodo
3. a destra



in profondità da sinistra a destra

percorso prefisso:

1. nodo
2. a sinistra
3. a destra

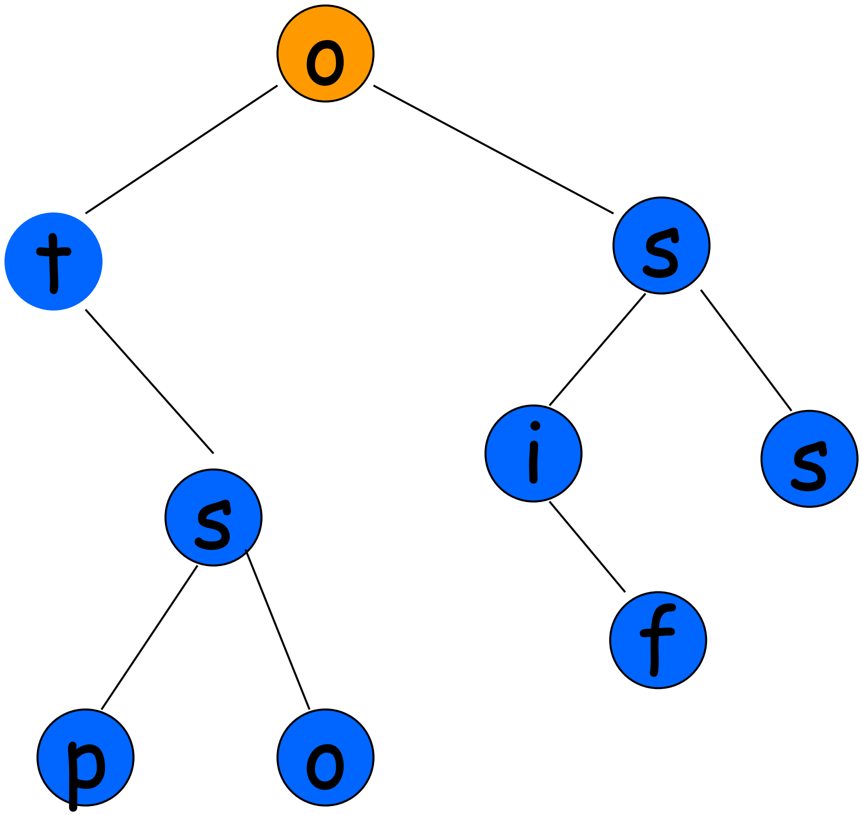


percorso postfisso:

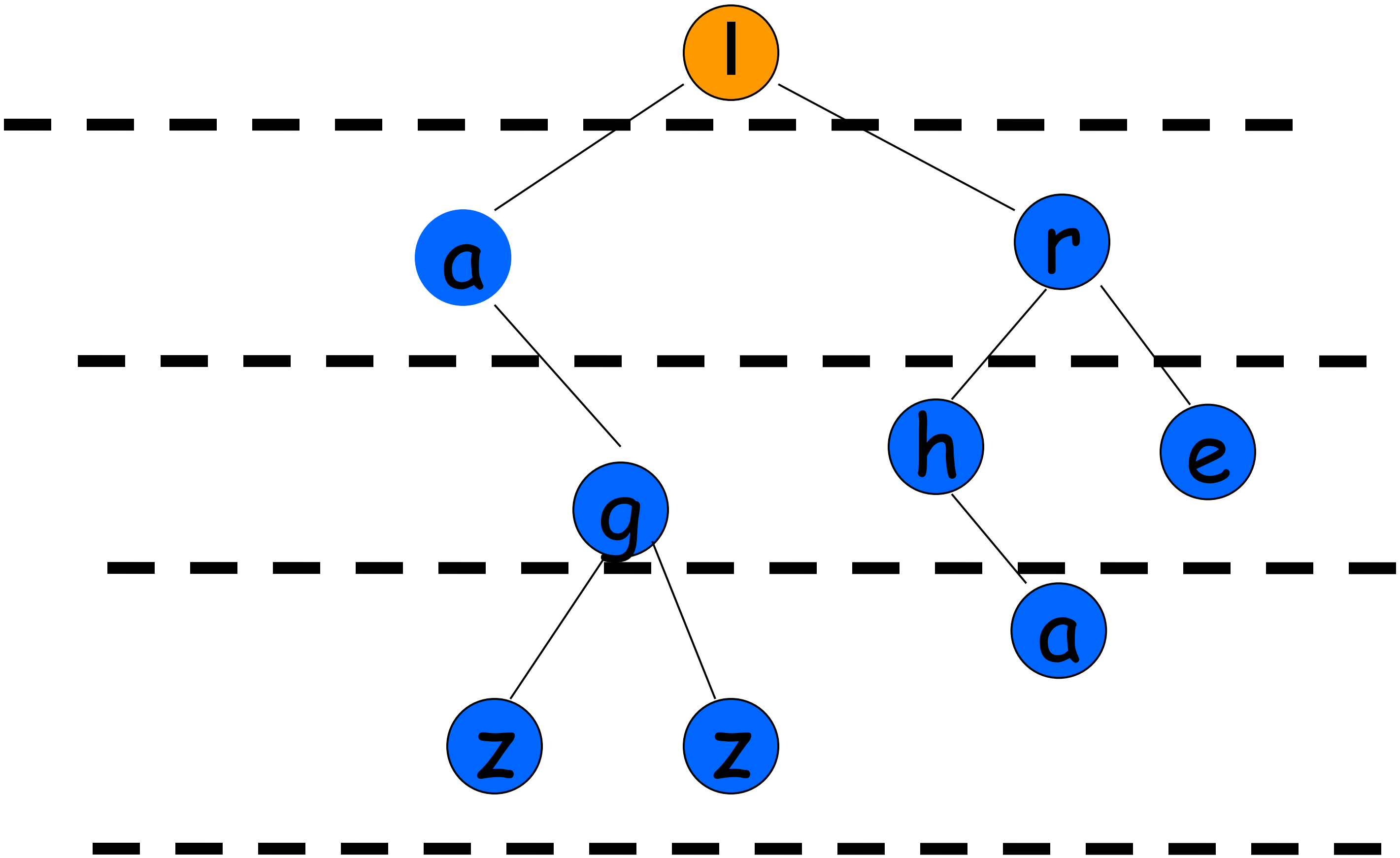
1. a sinistra

2. a destra

3. nodo

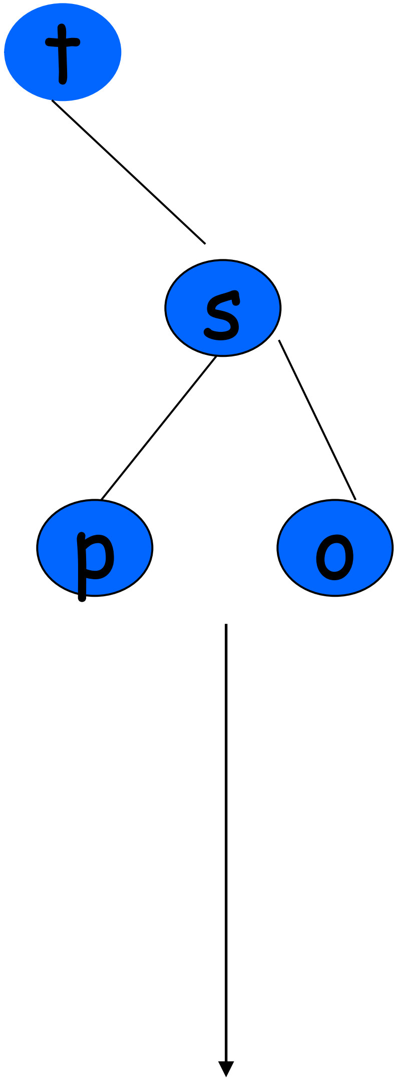


in larghezza



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;}

};
```



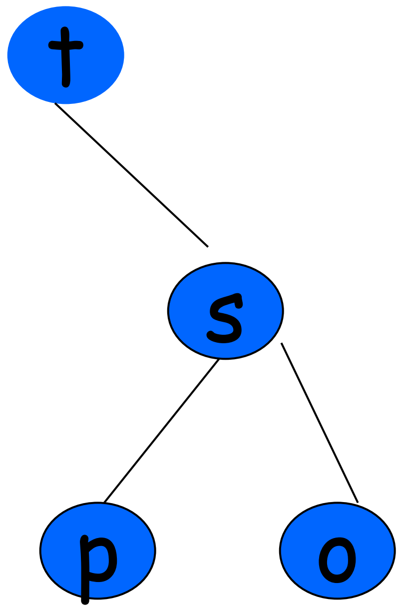
costruiamo questo albero:

```
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:

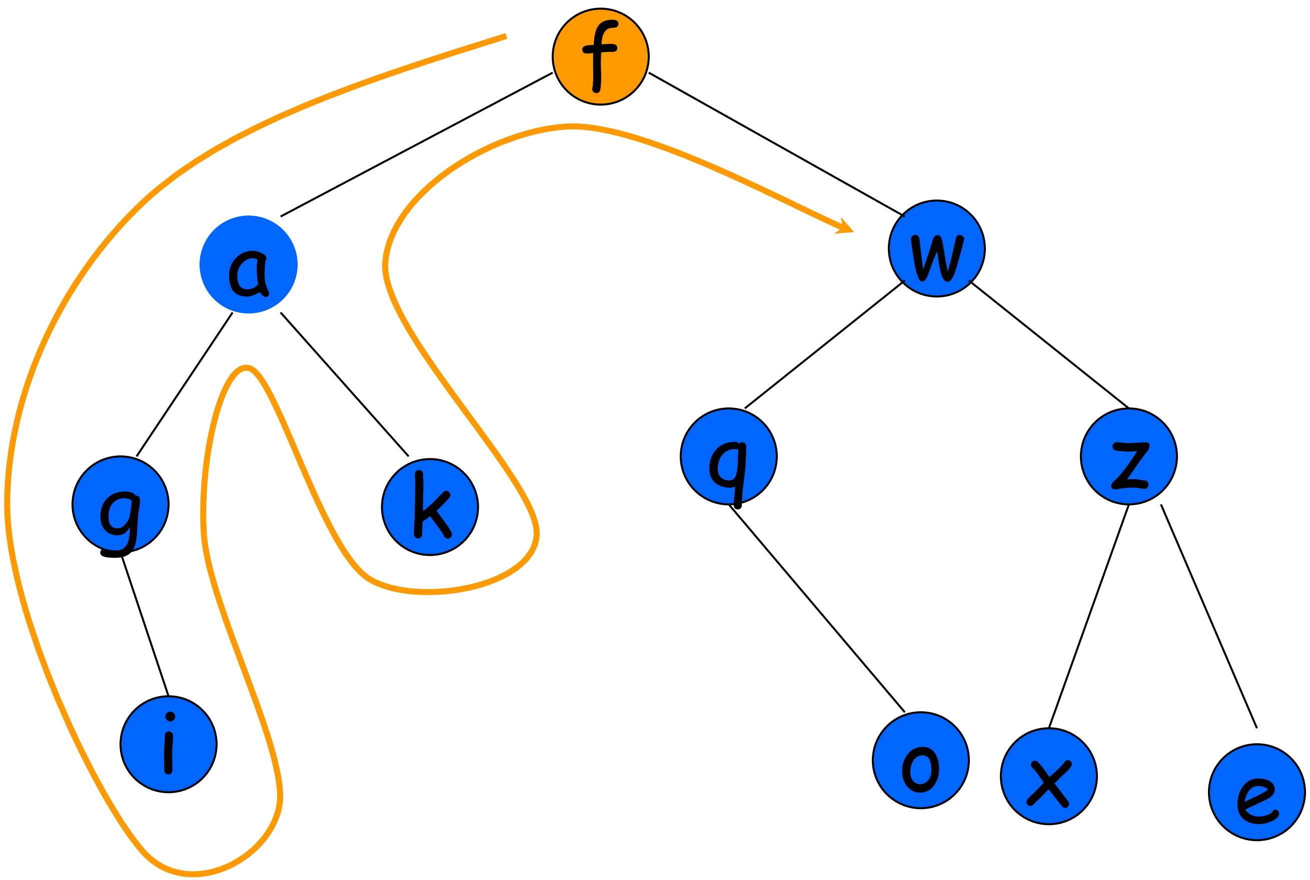
```
void infix(nodo *x){
if(x) {
infix(x->left);   // stampa albero sinistro
cout<<x->info; // stampa nodo
infix(x->right);  // stampa albero destro
}}

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);
}
```

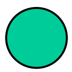
invocazione

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

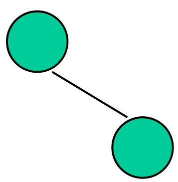


cerchiamo 'w'

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;
}}
```

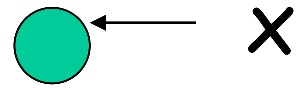
proviamo che è corretto:

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

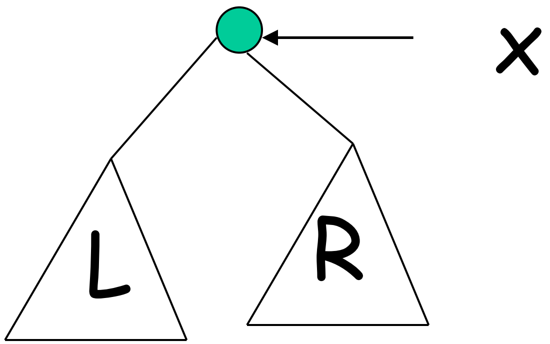
un solo nodo




```
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

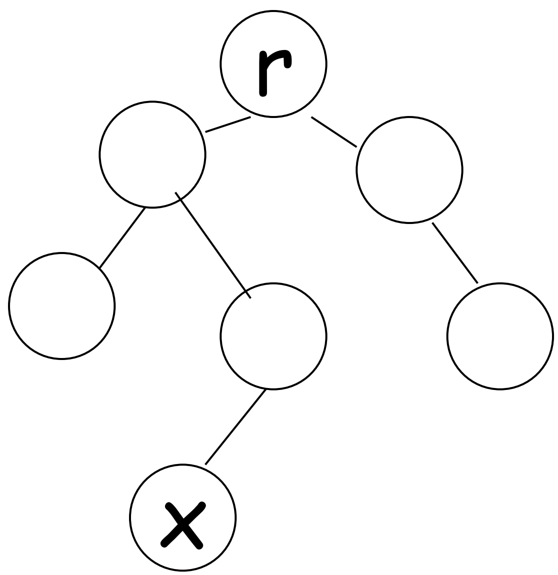
maggiore delle 2

+ 1 OK

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=[] elung =0

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

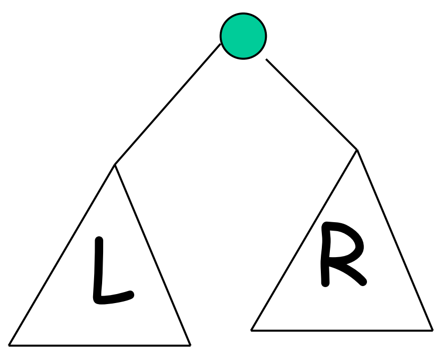
```
nodo * trova(nodo *x, int*  C, int lung)
{ if(!x) return 0;  // fallito
if(!lung) return x;  //trovato
if(*C==0)
return trova(x->left, C+1, lung-1);
else
return trova(x->right,C+1, lung-1);
}
```

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

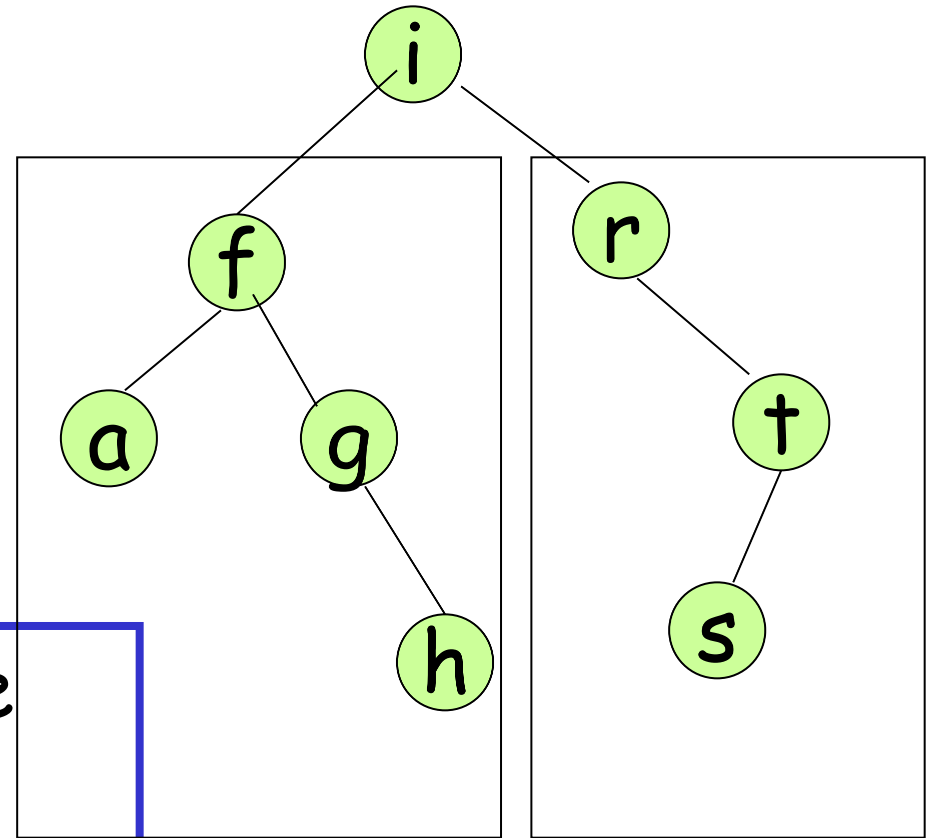
o l'unico nodo se l'albero era vuoto

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 2

binary search trees (BST):

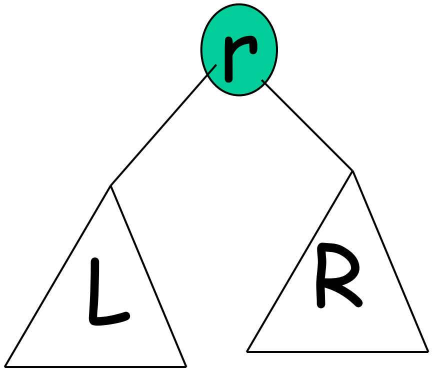


ogni nodo è maggiore dei nodi nel suo sottoalbero sinistro e minore di quelli del suo sottoalbero destro in un BST è facile (efficiente) trovare un nodo con un certo campo info y

e restituire il puntatore a quel nodo se lo troviamo

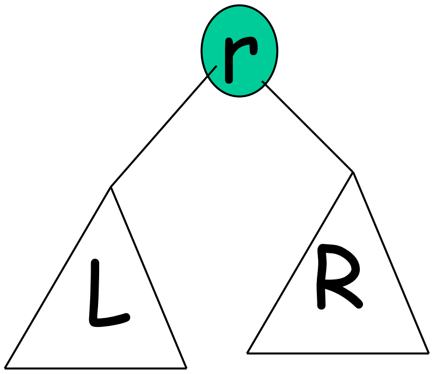
e 0 altrimenti

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 più semplice:



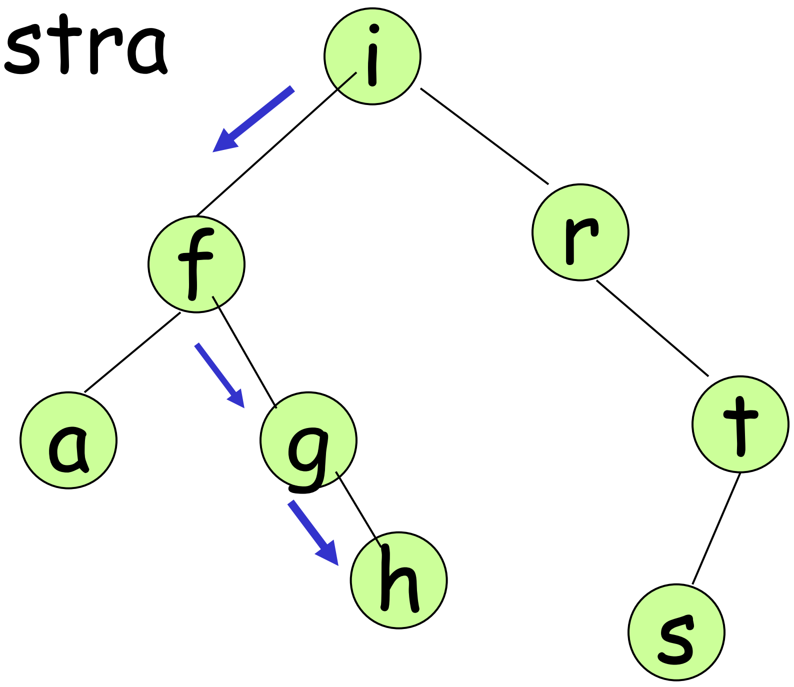
1. r→info == y restituisco r, altrimenti:
2. se r→info > y ➔ cerco solo in L
   altrimenti cerco solo in R

cerchiamo h:

h<i andiamo a sinistra

h>f destra

h>g destra



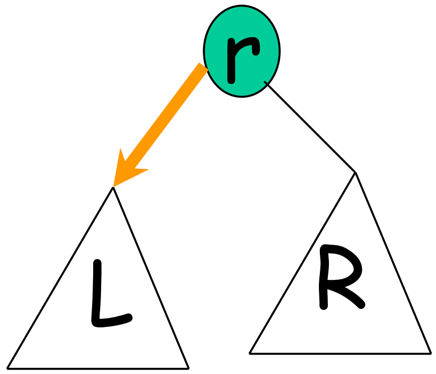
trovato !!!

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

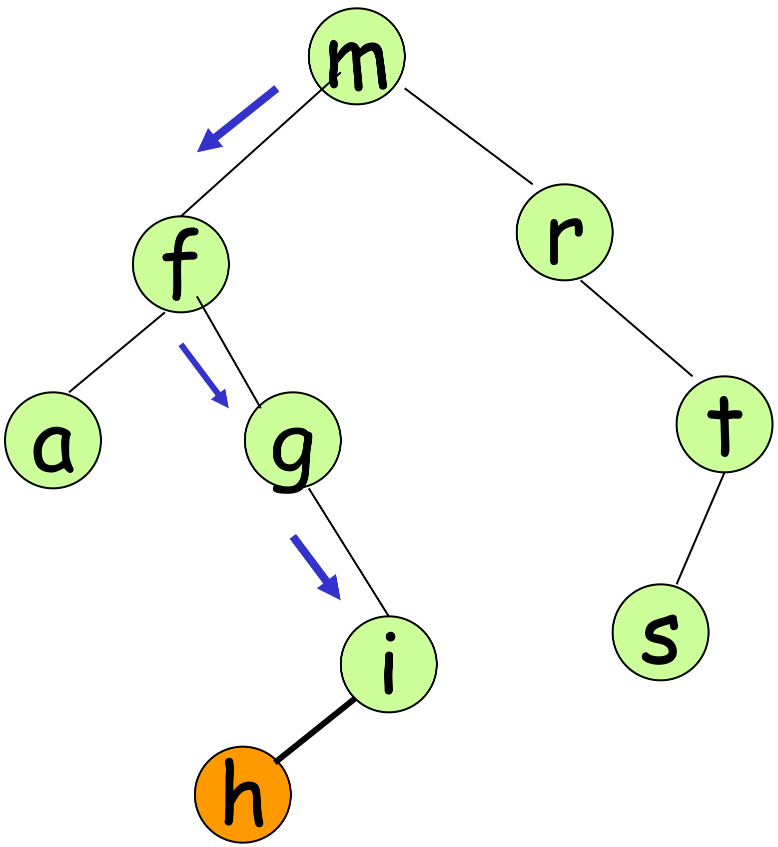


se l'albero è equilibrato, altezza = log n

dove n è il numero dei nodi dell'albero

una bella differenza tra n e log n !!!

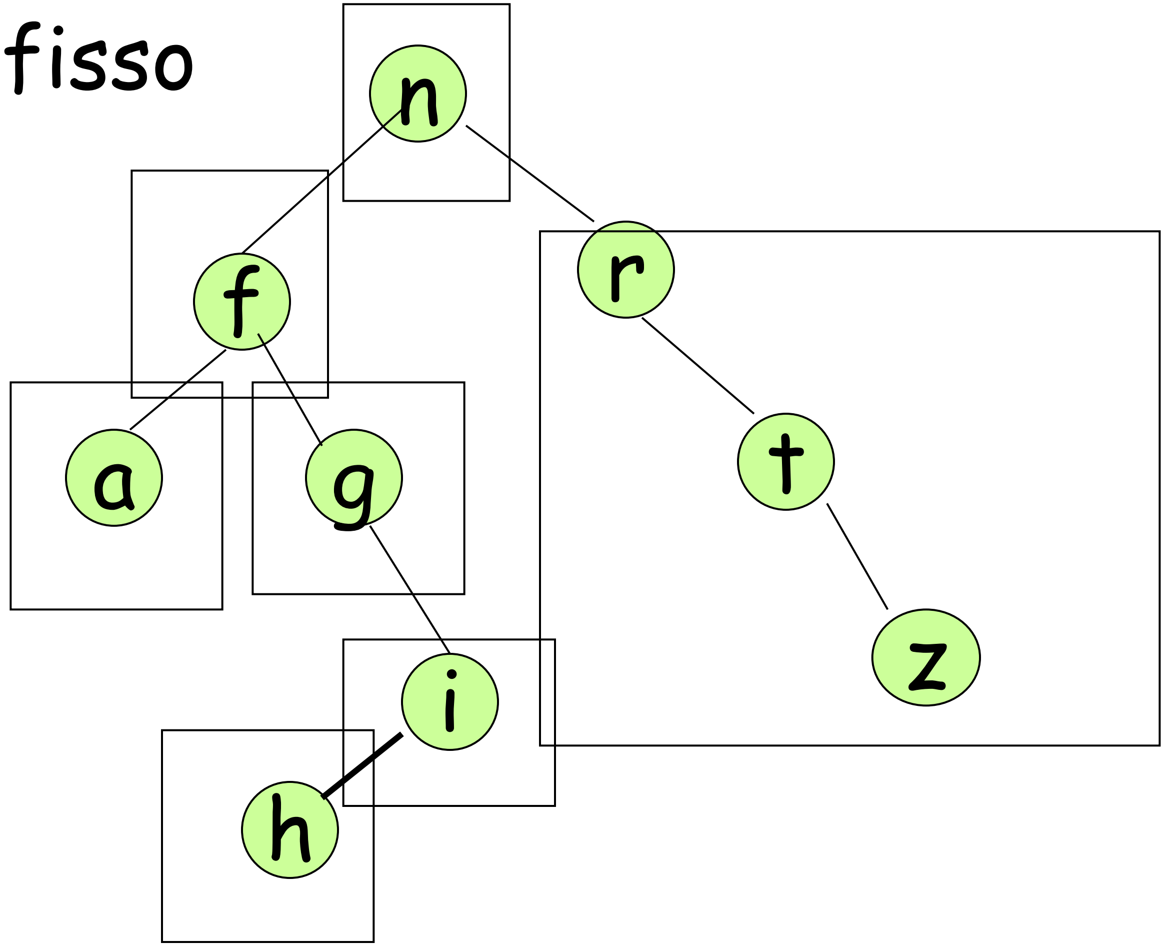
se h non ci fosse dove andrebbe inserito?



inserimento in un BST:

```
nodo * insert(nodo *r, nodo*n){
if(!r) return n;
if(r→info > n→info)
r→left=insert(r→left, n);
else
r→right=insert(r→right, n);
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

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

```
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 profondità minima tra le foglie

e poi vogliamo anche una foglia a profondità minima

usiamo:

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

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

        }
return meglio;}
```

vogliamo anche il puntatore al nodo:

la funzione restituisce un valore:

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

```
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==-1 || b.prof==-1)
                if(a.prof==-1) return b;
                else return a;
        else
                if(a.prof>b.prof) return b;
                else
                        return a;
        }
return foglia(0,-1);
}
```

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(m.prof!=-1 && prof>=m.prof) return;
if(x)
        {
          if(leaf(x))
                {m.fo=x; n.prof=prof; return;}
          else
                {

                        f(x->left,prof+1,m);

                        f(x->right,prof+1,m);
                }

        }
}
```

invocazione: foglia p(0,-1);
f(root,0,p);

restituire un nodo che dista k da una foglia trovare un nodo a distanza k da una foglia   VARI MODI

uno è di esaminare ogni nodo r e di invocare da r una funzione ricorsiva che risponde true sse trova una foglia a distanza k da r

se la risposta è true allora la funzione restituisce r, altrimenti prova con i suoi figli

caso base: albero vuoto: return false;

```
bool G(nodo *x, int k)
{if(x)
{if(leaf(x) && k==0) return true;
return G(x->left,k-1) || G(x->right,k-1);}
else
return false;
}
```

si può fare meglio?

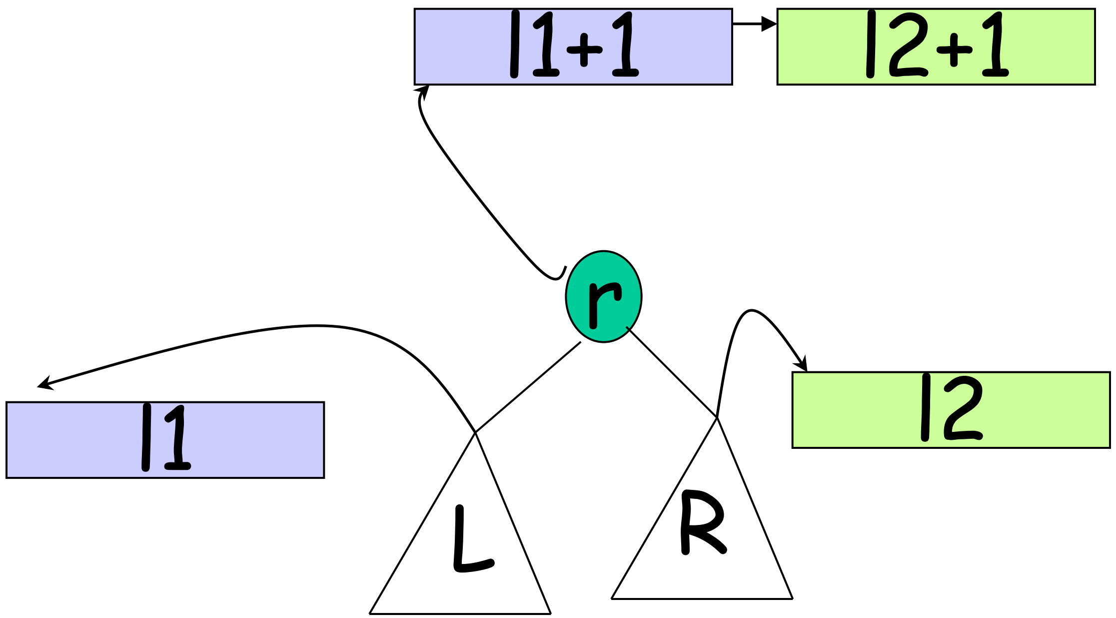
```
nodo* F(nodo* x, int k)
{if(x)
{if(G(x,k)) return x;
nodo* y=F(x->left,k);
if(y) return y;
return F(x->right,k);
}
else
return 0;
}
```

un'altra soluzione è di aggiungere ad ogni nodo la lista delle sue distanze dalle foglie:

1. con questa informazione è facile percorrere l'albero ed in ogni nodo rispondere se soddisfa la condizione o no

2. date le liste dei figli sinistro e destro di un nodo, basta concatenarle per ottenere quella del padre

caso ricorsivo:



caso base ? foglia !

2 tipi nodo:

per le liste e per gli alberi

```
struct nodoL{int dist; nodoL* next;};

struct nodoA {nodoL* lista;
nodoA* left,*right;};
```

dato un albero binario decoriamo ogni nodo con le liste delle sue distanze dalle foglie

```
void F(nodoA* R)
{if(R)
{if( leaf(R) )
R→lista=new nodoL(0,0);
else
{nodoL* L1=0, *L2=0;
F(R→left); F(R→right);  // NB
if(R→left)
L1=copia(R→left→lista);
if(R→right)
L2=copia(R→right→lista);
add(L1,1); add(L2,1);
R→lista=concatena(L1,L2);}}}
```

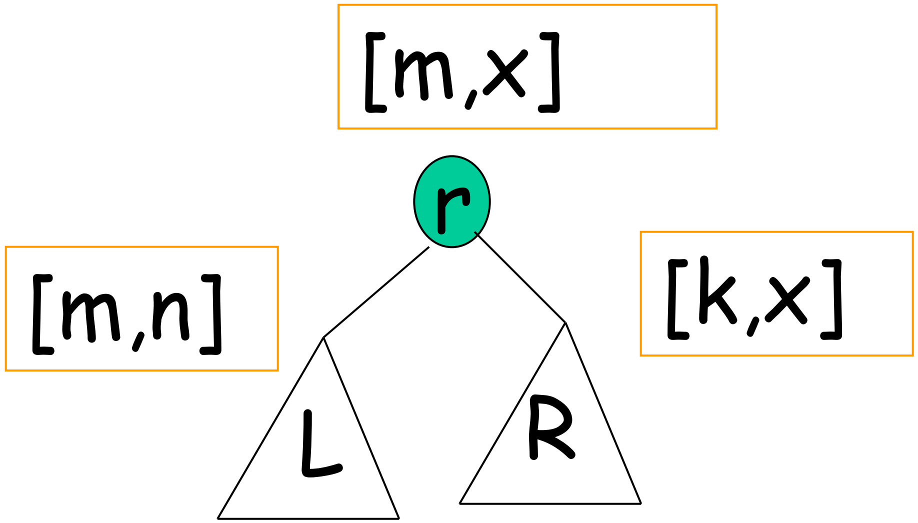
```
nodoL* copia(nodoL* x)
{if(x)
{

nodoL* y=new nodoL(x->dist);
y->next=copia(x->next);
return y;
}
else
return 0;
}
```

```
nodoL * concatena(nodoL* x, nodoL* y)
{ if(x)
        {
        x->next=concatena(x->next);
        return x;
        }
else
        return y;
}
```

fate voi: add(nodoL* x, int k)

ricerca + efficiente

BST: aggiungere ad ogni nodo n l'intervallo [l1, l2] dei valori contenuti nell'albero radicato in n




```
struct vallo {int primo, secondo;};
```

i nodi dell'albero hanno tipo:

```
struct nodoA{int info; vallo v; nodo * left,
* right};
```

void F(nodo *R)

```
{
if(R)
{F(R→left); F(R→right); //NB
vallo i1=vallo(R→info, R→info), i2=.....;
if(R→left)
i1=R→left→v;
if(R→right)
i2=R→right→v;
R→v=vallo(i1.primo,i2.secondo);
}
```

la ricerca è + efficiente

```
nodo * ric(nodo* r, int y)
{if(!r || (y<r->vallo.primo || y>r->vallo.secondo)))
return 0;
if(y==r->info) return r;
if(y<r->info)
        return ric(r->left,y);
else
        return ric(r->right,y);
}
```

è importante che

le operazioni conservino le informazioni nei nodi:

inserimento che mantiene gli intervalli

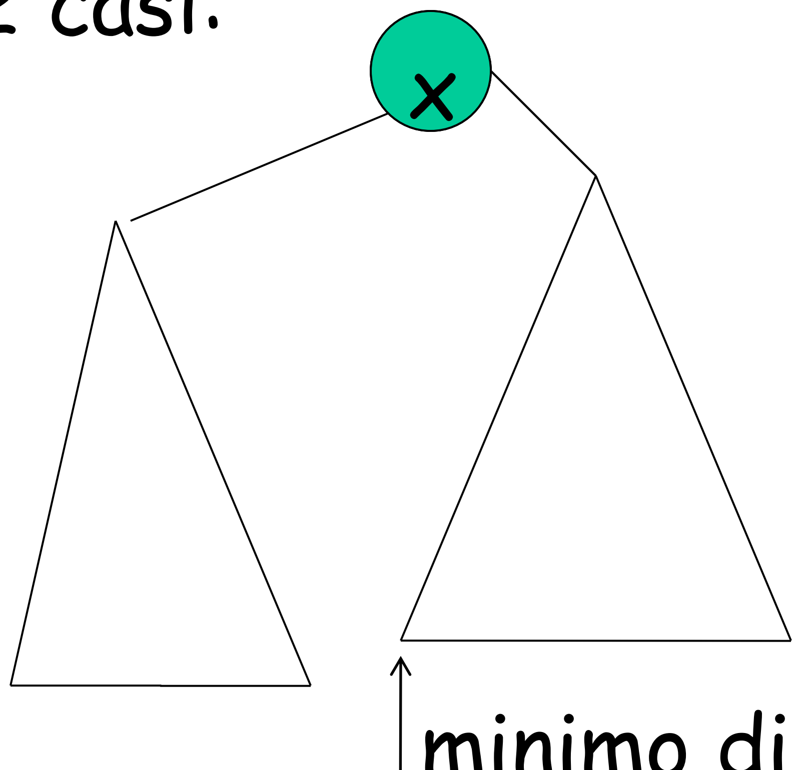
```
nodoA * insert(nodoA *r, nodoA*n){
if(!r) return n;
if(r→info > n→info)
{
        if((r->v). primo > n->info)
                (r->v).primo=n->info;
        r→left=insert(r→left, n);
}
 else
{
if((r->v).secondo<n->info)
        (r->v).secondo=n->info:
r→right=insert(r→right, n);
}
return r; }
```

in caso si debba risalire l'albero e non solo scenderlo, è necessario inserire in ogni nodo anche il puntatore al padre:

```
void padre(nodo*x, nodo*p) {
if(!x) return ;
x->padre =p;
padre(x->left,x);
padre(x->right,x);
}
invocazione padre(root, 0),
```

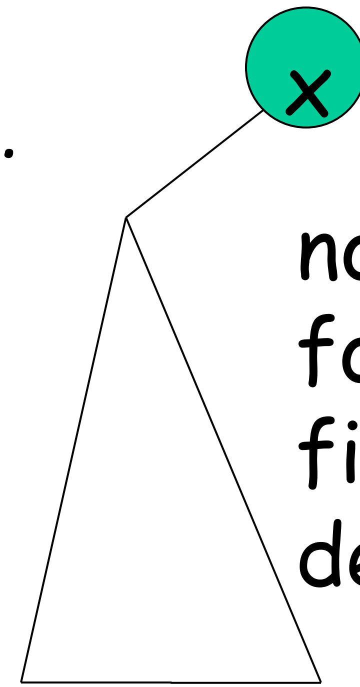
successore di un nodo n in un BST= nodo stampato subito dopo n

2 casi:

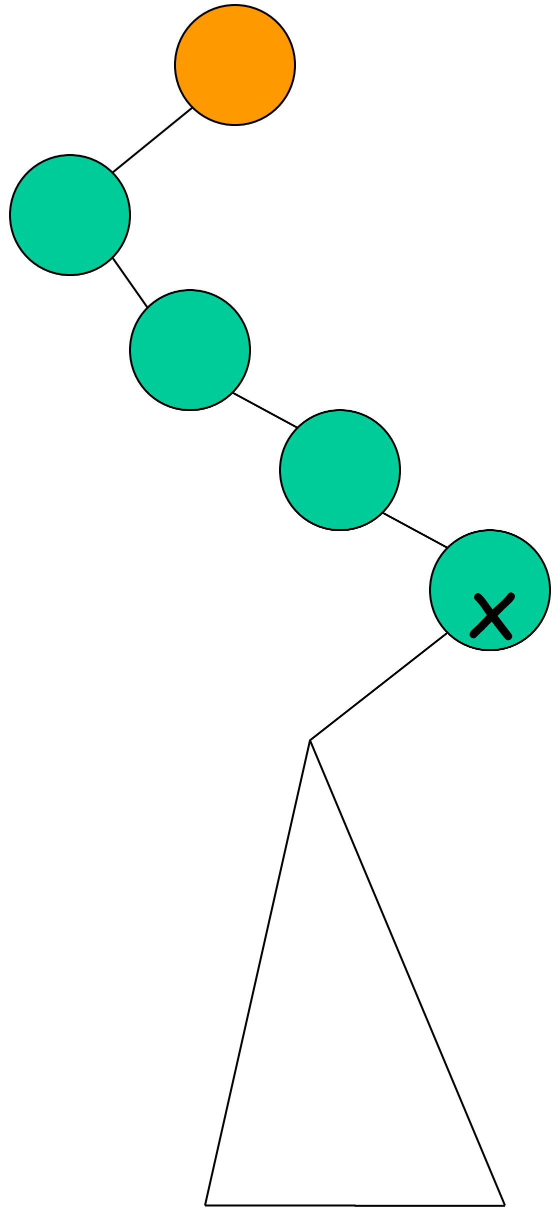


minimo di quest'albero

ma se......



non ci fosse il figlio destro ?



si deve risalire
verso la radice

fino a quando?

o trovo la radice

o vado a destra

successore iterativo

```
nodo *succ(nodo*x) {
if(x->right) return min(x->right);
nodo *y=x->padre;
while(y && y->right==x)
{x=y; y=y->padre;}
return y;
}
```

successore ricorsivo

```
nodo *f(nodo*y, nodo*x)  {
if(! y || y->left==x) return y;
return f(y->padre,y);
}
nodo *succ(nodo*x) {
if(x->right) return min(x->right);
return f(x->p,x);
}
```

come ragionare

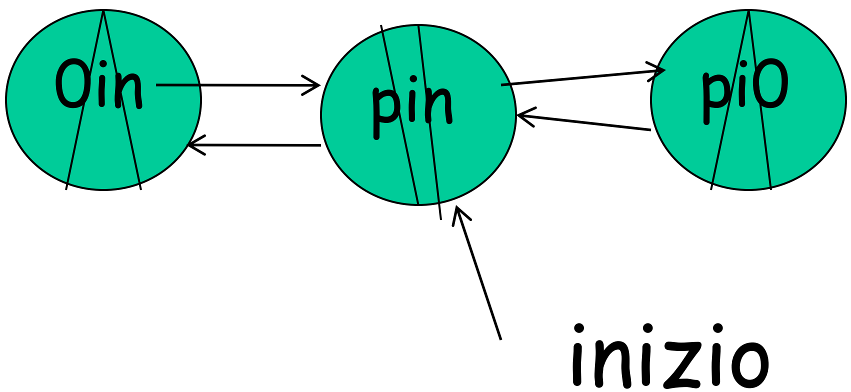
capire se dobbiamo cercare in uno solo dei sottoalberi oppure in entrambi: *sempre uno solo, sempre entrambi, o dipende*

determinare il percorso dell'albero

ci sono calcoli evitabili ?

distinguere i casi base

liste doppie



esercizi
inserimento/cancellazione
inserire a sinistra a destra

cercare il successore di un nodo in un BST: