# In questa lezione

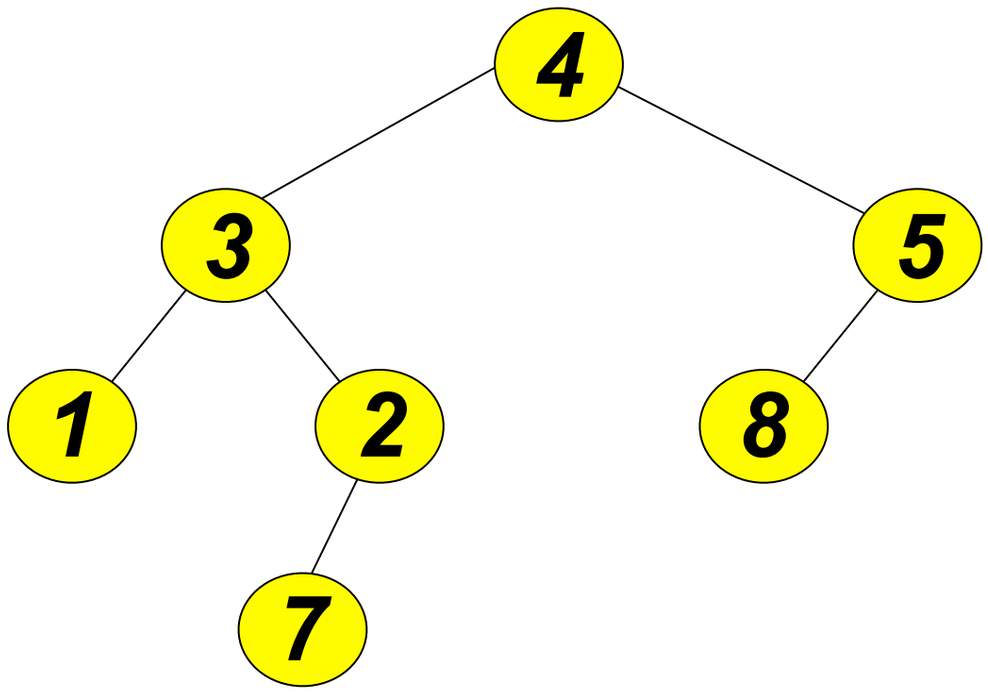
- **Alberi binari:**
  - **visite e esercizi su alberi binari**

- **[CLRS09] cap. 12 per la visita inorder**

# Visita inordine di un albero binario

**visita inordine(x)**
**se l'albero x non è nullo allora**
**visita inordine il sottoalbero sinistro di x**
**visita x**
**visita inordine il sottoalbero destro di x**

**Input:**




**Output: 1372485**

# Rappresentazione in memoria alberi binari

**Dato un albero binario t.**

**Nella rappresentazione in memoria assumiamo che**

**ogni nodo di t abbia**

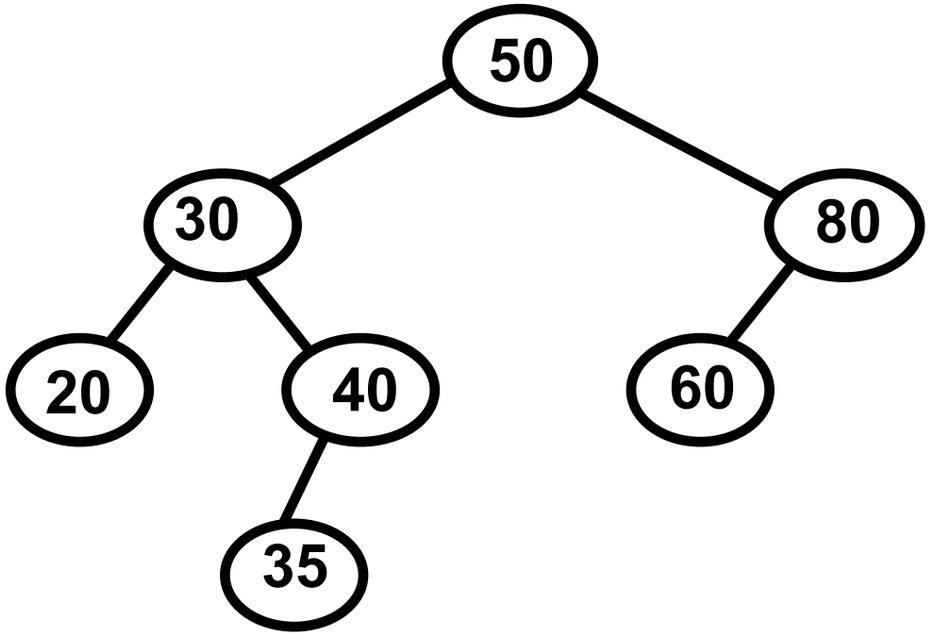
**un campo key, che contiene la chiave del dato**

**un campo left, con un puntatore al figlio sinistro,**

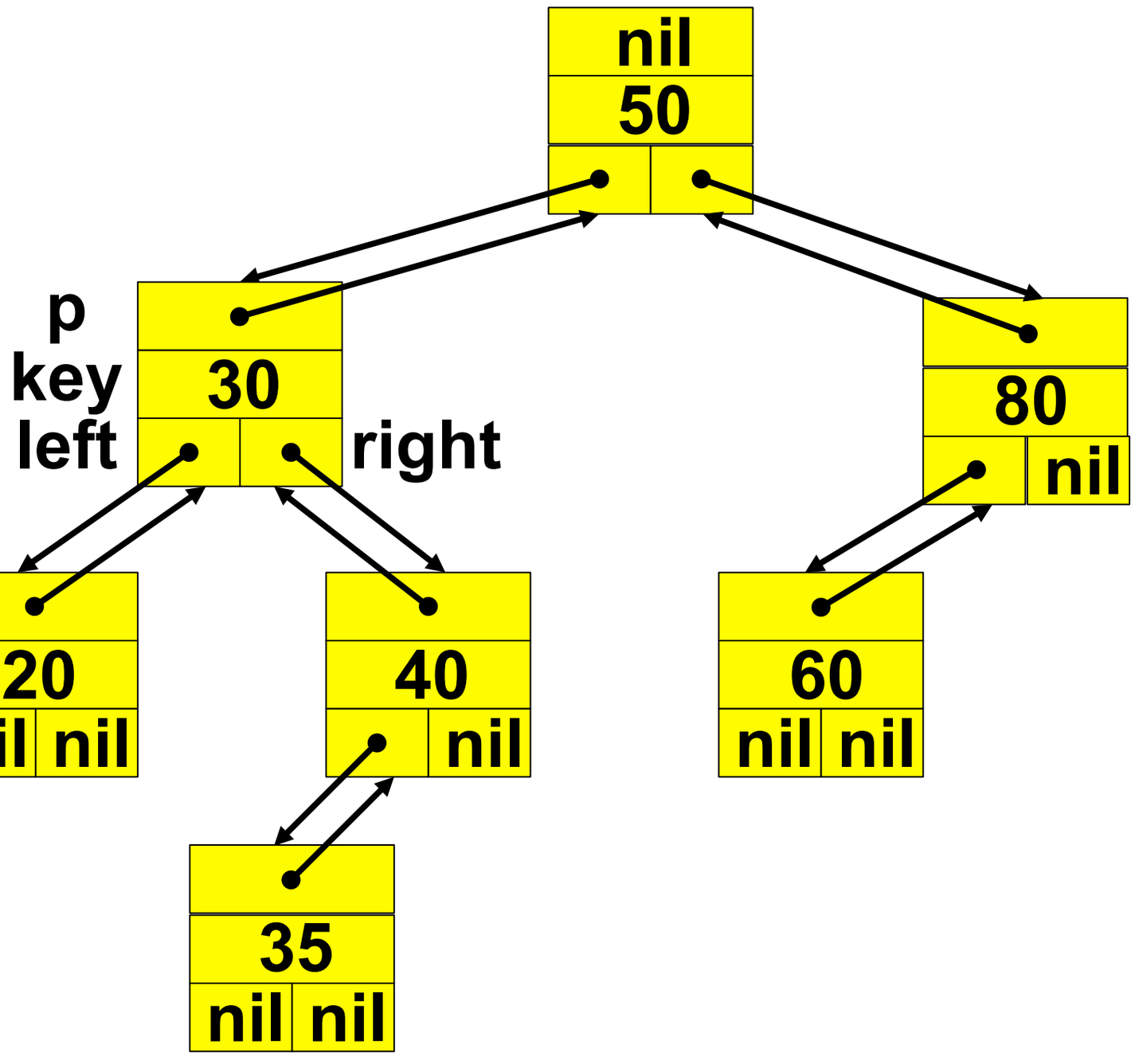
**un campo right, con un puntatore al figlio destro**

**un campo p, con un puntatore al padre**

# Rappresentazione in memoria



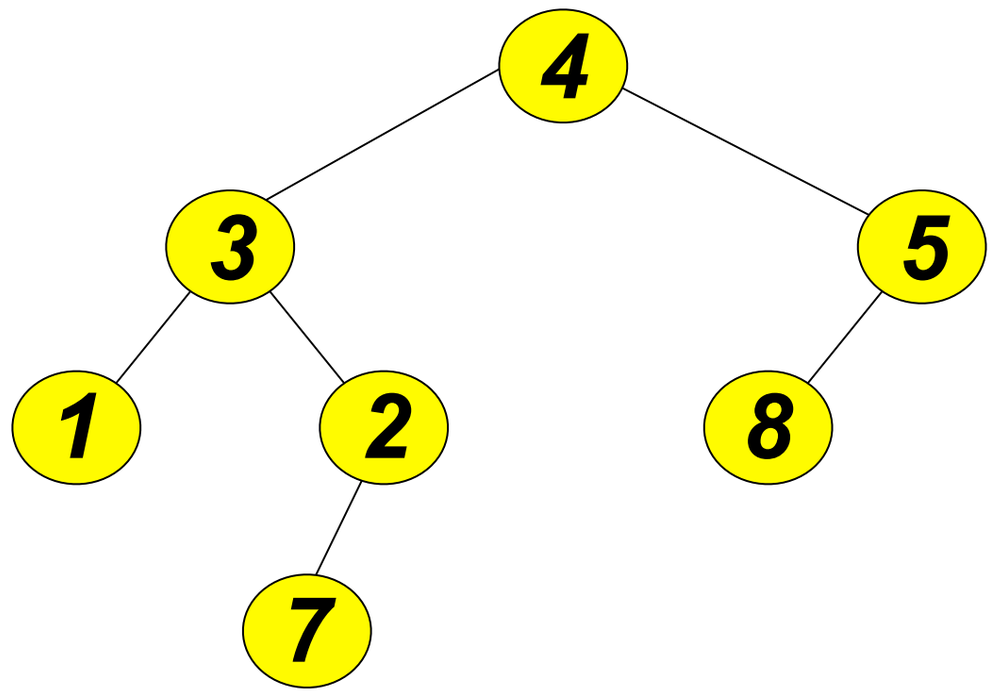

rappresentazione



# Visita inordine di un albero binario

```
Inordine(x)
 if x ≠ nil then
      Inordine(left[x])
      print   key[x]
      Inordine(right[x])
```

Input:




Inordine(4)

Inordine(3)

Inordine(1)

Inordine(nil)

print key(1)

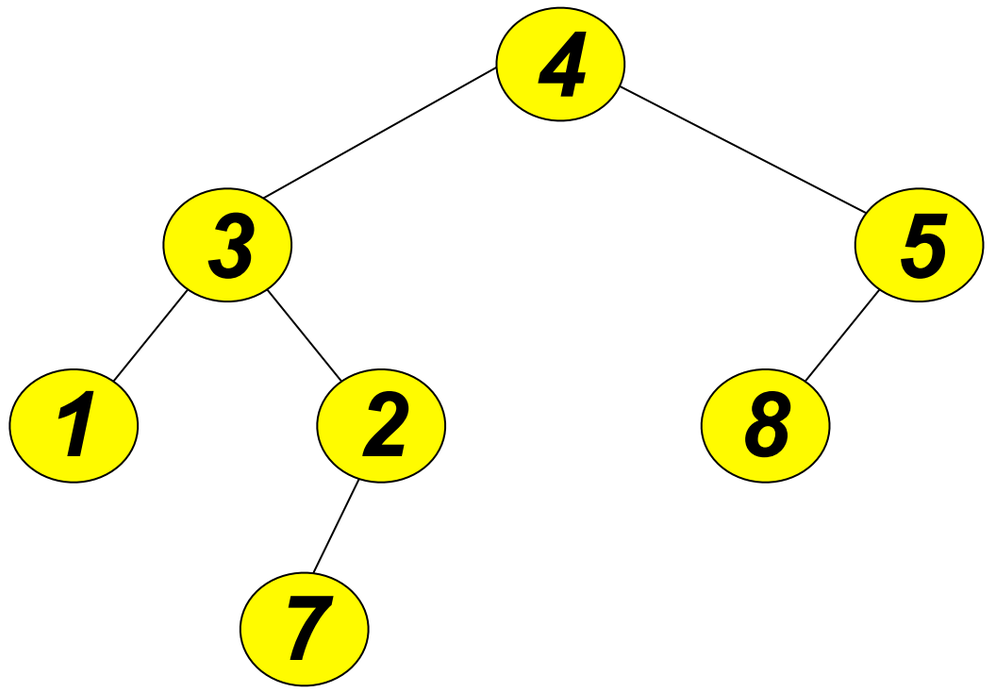
print key(3)

Output: 1 3

# Visita inordine

```
Inordine(x)
 if x ≠ nil then
      Inordine(left[x])
      print   key[x]
      Inordine(right[x])
```

Input:




Inordine(4)

Inordine(3)

Inordine(2)

Inordine(7)

print key(7)

print key(2)

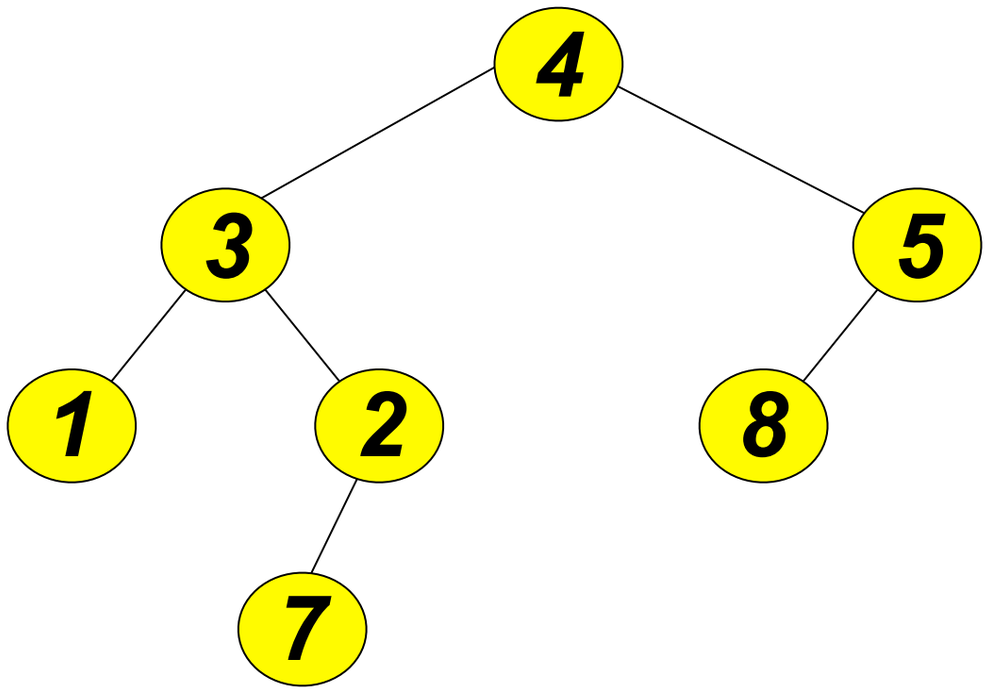
print key(4)

**Output: 1 3 7 2 4**

# Visita inordine

```
Inordine(x)
 if x ≠ nil then
      Inordine(left[x])
      print   key[x]
      Inordine(right[x])
```

Input:




```
Inordine(4)
    Inordine(5)
    Inordine(8)
print key(8)
print key(5)
```

**Output: 1 3 7 2 4  8  5**

# Complessità visita inordine

**Sia n il numero dei nodi dell'albero binario e m quello del suo sottoalbero sinistro allora la relazione di ricorrenza è**

$T(0) = c$
$T(n) = T(m) + T(n-m-1) + d \quad (c,d > 0)$

```
Inordine(x)
 if x ≠ nil then
     Inordine(left[x])
     print key[x]
     Inordine(right[x])
```

**Ipotizziamo che T(n) = O(n).**
**Verifichiamo che T(n) ≤ kn, per una costante k>0, a partire da un certo n in poi.**

$T(n) = T(m) + T(n-m-1) + d$
$\leq km + k(n-m-1) + d$
$= km + kn - km - k + d = kn - k + d$

**si vuole che $kn - k + d \leq kn$ questo è vero per $k \geq d$.**
**Guardiamo al caso base $T(1) = T(0) + T(0) + d = 2c+d$ e scegliamo quindi $k = 2c+d$, per concludere che $T(n) = O(n)$ perché abbiamo trovato una costante k tale che $T(n) \leq kn$ per ogni $n \geq n_0 = 0$.**

**Quindi T(n) = O(n)**

# Complessità visita inordine

```
Inordine(x)
 if x ≠ nil then
      Inordine(left[x])
      print key[x]
     Inordine(right[x])
```

**T(n) = Ω(n), perchè tutti i nodi sono visitati.**

**T(n) = Θ(n)**

# Altre visite

```
Inordine(x)
 if x ≠ nil then
      Inordine(x.left)
      print key(x.right)
      Inordine(x.right)
```

```
Postorder(x)
 if x ≠ nil then
   Postorder(x.left)
   Postorder(x.right)
   print key[x]
```

definiamo altre due visite  modificando l'ordine di visita della radice rispetto ai suoi sottoalberi,

Visita postOrder :
visita il sottoalbero sinistro
visita il sottoalbero destro
visita la radice

Visita preordine :
visita la radice
visita il sottoalbero sinistro
visita il sottoalbero destro

```
Preordine(x)
if x ≠ nil then
  print key[x]
  Preordine(x.left)
  Preordine(x.right)
```

$T(n) = \Theta(n)$

# Altri conteggi: numero dei nodi

```
nNodi(x)

input: x è il puntatore alla radice di un
albero binario

output: il numero dei nodi dell'albero
di radice x
if x == nil then return 0
n1 = nNodi(x.left)
n2 = nNodi(x.right)
n = n1 + n2 + 1
return n
```

**E' una visita postorder in cui la "visita" di un nodo è effettuata calcolando la somma dei valori ottenuti dalle visite dei due sotto alberi. Il tempo di esecuzione dell'algoritmo per il calcolo del numero dei nodi, n, è θ(n).**

# Altri conteggi: numero dei nodi

**nNodi(x)**

**input: x è il puntatore alla radice di un albero binario**

**output: il numero dei nodi dell'albero di radice x**

```
if x == nil then return 0
return nNodi(x.left) + nNodi(x.right) +1
```

**La versione più snella.**

# Altri conteggi: altezza

```
Altezza(x)
input: x è il puntatore alla radice di un albero binario
output: l'altezza dell'albero di radice x

if x == nil then return -1
h1 = Altezza(x.left)
h2 = Altezza(x.right)
h =  max{h1,h2} +1
return h
```

**E' una visita postorder in cui la "visita" di un nodo è effettuata calcolando l'altezza in base ai valori ottenuti dalle visite dei due sotto alberi. Il tempo di esecuzione dell'algoritmo per il calcolo dell'altezza, in un albero con n nodi, è θ(n).**

# Altri conteggi: altezza 2

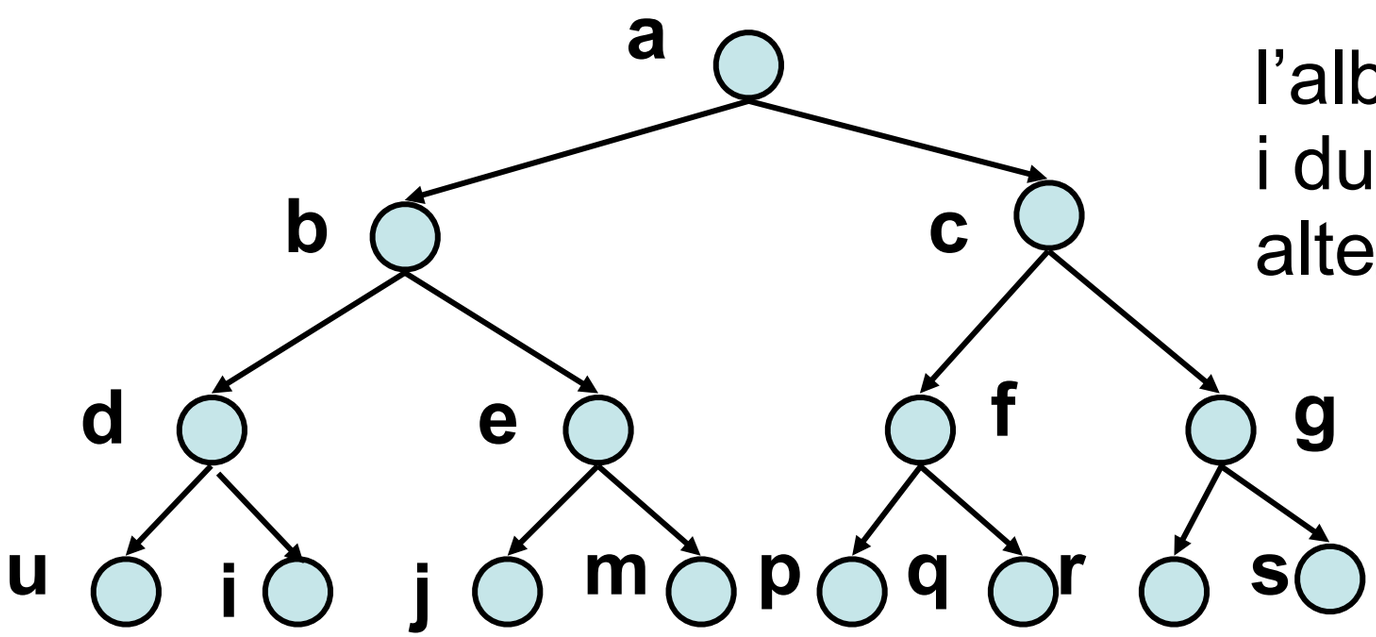
```
Altezza(x)
input: x è il puntatore alla radice di un albero binario
output: l'altezza dell'albero di radice x

if x == nil then return -1
return max{Altezza(x.left),Altezza(x.right)} +1
```

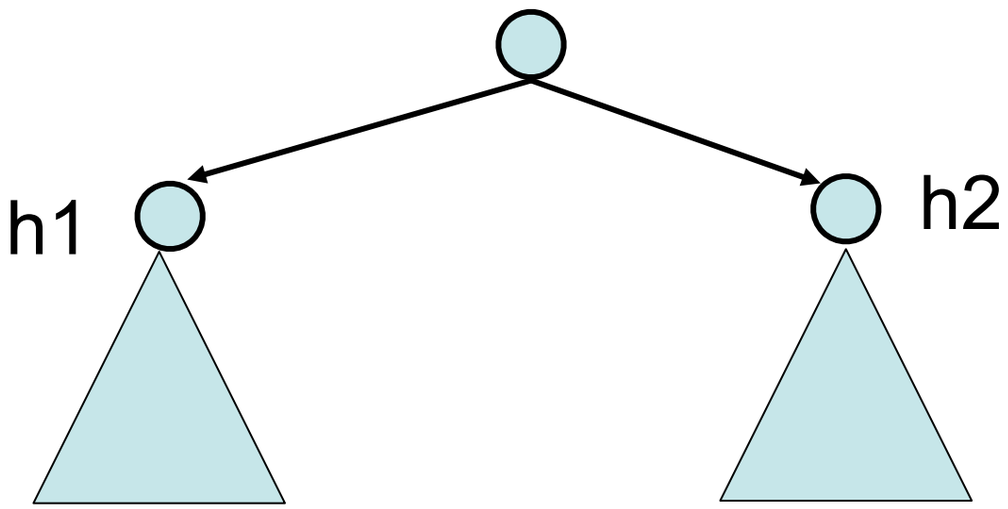
**Versione più sintetica e veloce!**

# Esercizio 2

**Si costruisca un algoritmo ricorsivo che verifica se un albero binario t, rappresentato con strutture a puntatori, è completo.**




l'albero è completo sse in tutti i nodi i due sottoalberi hanno la stessa altezza




se h1=h2 e i due sottoalberi sono completi restituiamo h1+1
altrimenti ?
prendiamo -2 che non è un'altezza.

# Esercizio 2

**Si costruisca un algoritmo ricorsivo che verifica se un albero binario t, rappresentato con strutture a puntatori, è completo.**

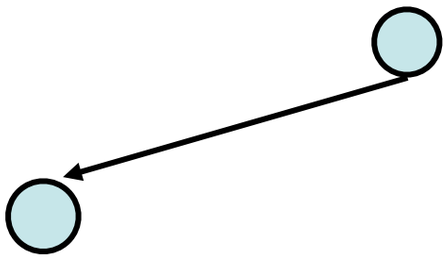
```
VerCompl(T)
input: un albero binario T, rappresentato con strutture a puntatori
postc: restituisce l'altezza dell'albero se l'albero è completo, -2
       altrimenti.
if (T==NIL) return -1
h1=VerCompl(T.left)
h2=VerCompl(T.right)
if (h1=-2) or ( h2=-2) then return -2  //almeno uno non è completo
        else if (h1=h2) then return h1+1
                        else return -2
```

**É una visita in postordine quindi VerCompl ha tempo di esecuzione asintotico in Θ(n), se n è il numero dei nodi dell'albero binario**

# Esercizio 3

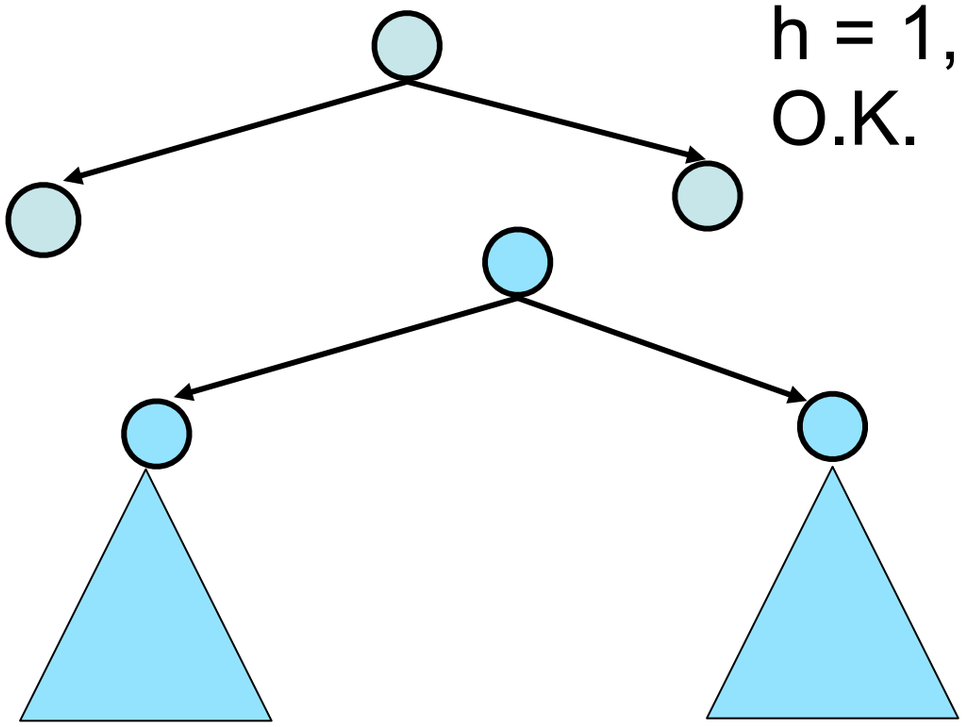
**Si costruisca un algoritmo ricorsivo che verifica se un albero binario t, rappresentato con strutture a puntatori, è quasi completo.**

∅ h= -1



h = 0, i due sottoalberi hanno altezza uguale -1

h = 1, il sinistro ha altezza h1=0 e il destro h2= -1, O.K.



h = 1, il sinistro ha altezza h1=0 e il destro h2 = 0, O.K.

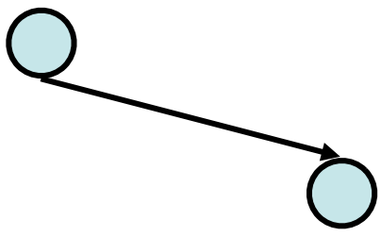
generalizzando è quasi completo se i due sottoalberi sono completi e hanno altezze h1=h2 oppure h2+1=h1

# Esercizio 3

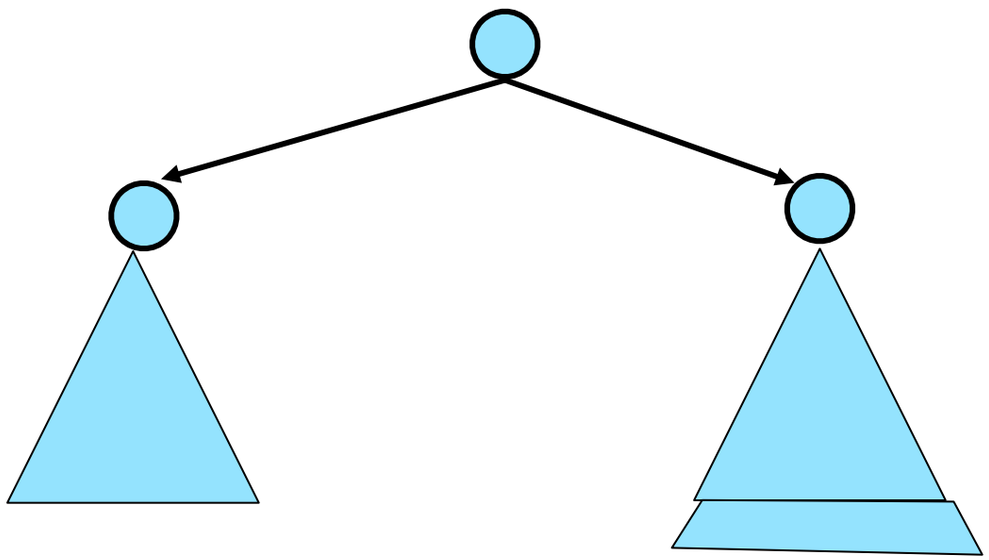
**Si costruisca un algoritmo ricorsivo che verifica se un albero binario t, rappresentato con strutture a puntatori, è quasi completo.**

∅ h= -1

h= 0, i due sottoalberi hanno altezza uguale -1



h = 1, il sinistro ha altezza h1=-1 e il destro h2= 0,
NO

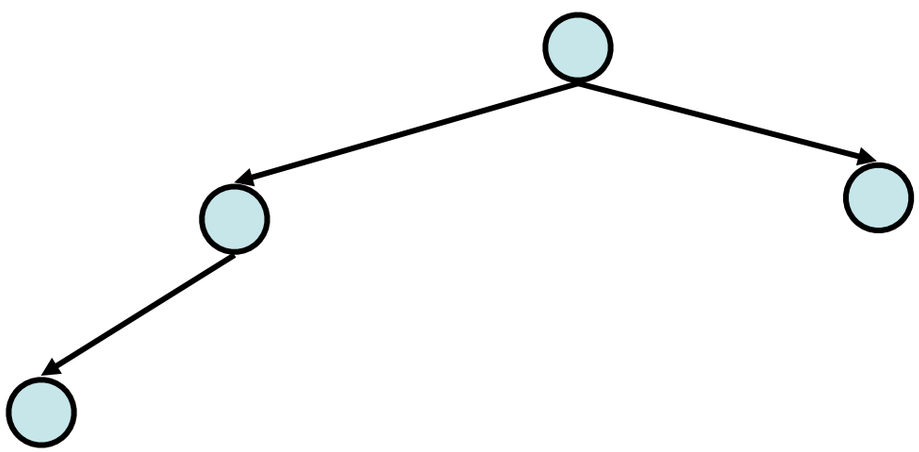


in generale se i due sottoalberi sono completi e h2= h1+1

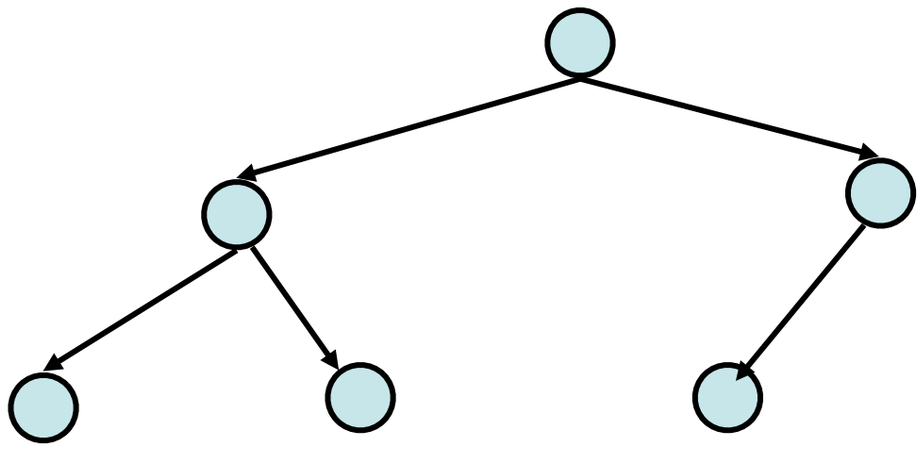
NO

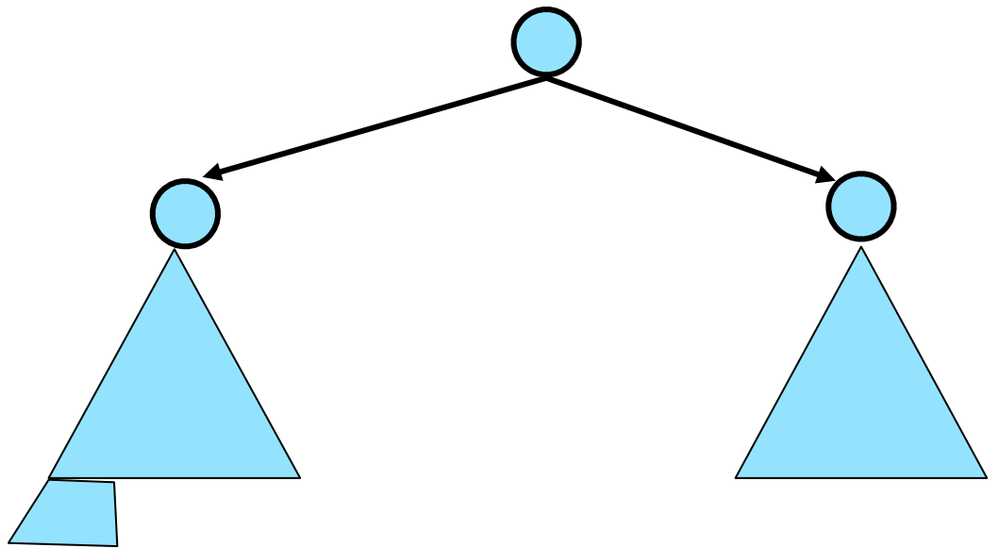
# Esercizio 3

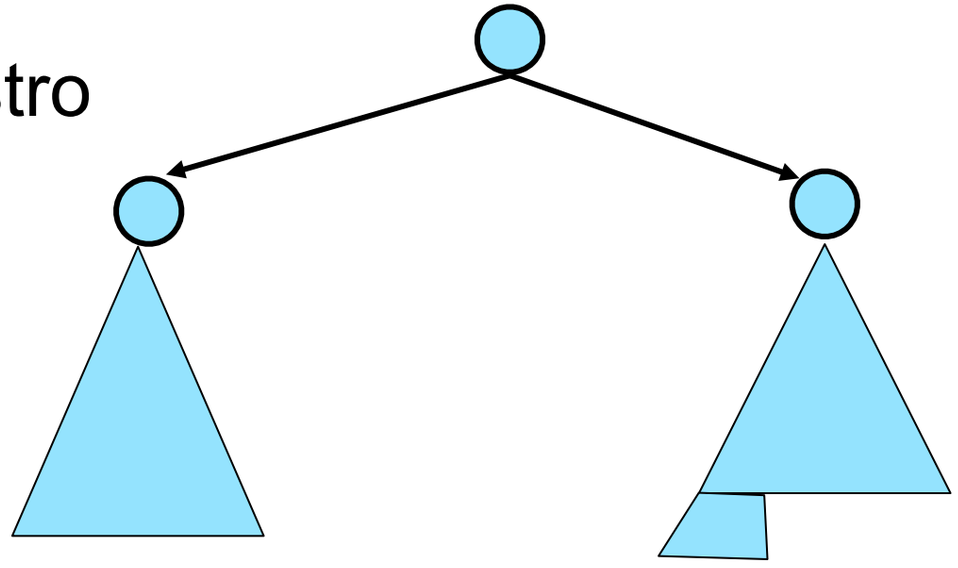
**Si costruisca un algoritmo ricorsivo che verifica se un albero binario t, rappresentato con strutture a puntatori, è quasi completo.**



si generalizza al caso in cui il sinistro è quasi completo e il destro completo e h1= h2+1, O.K.



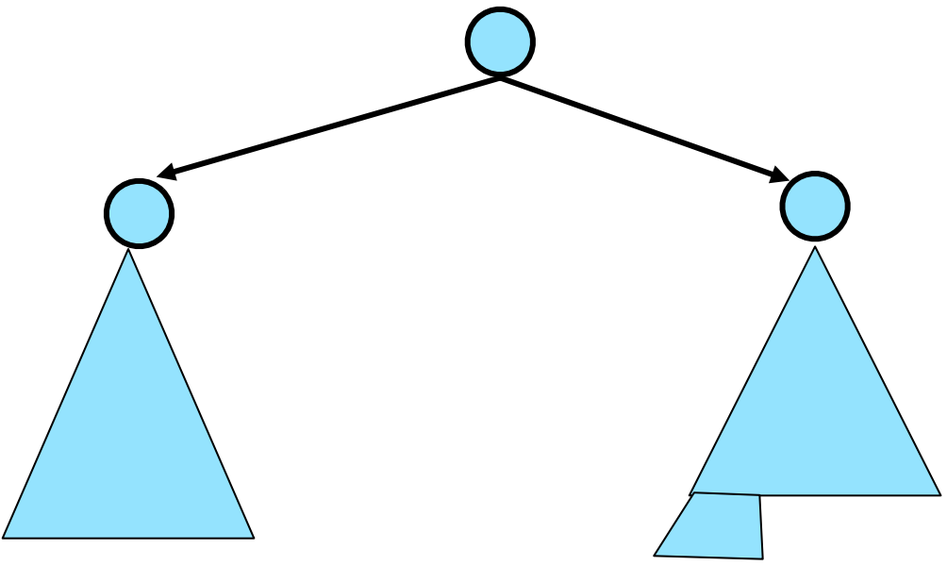




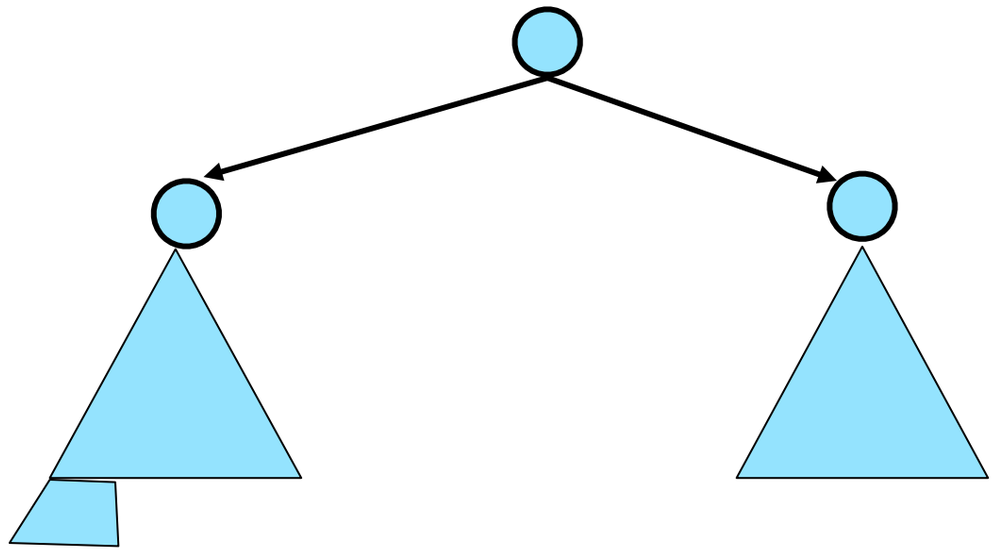
si generalizza al caso in cui il sinistro è completo e il destro quasi completo e h1= h2, O.K.

# Esercizio 3

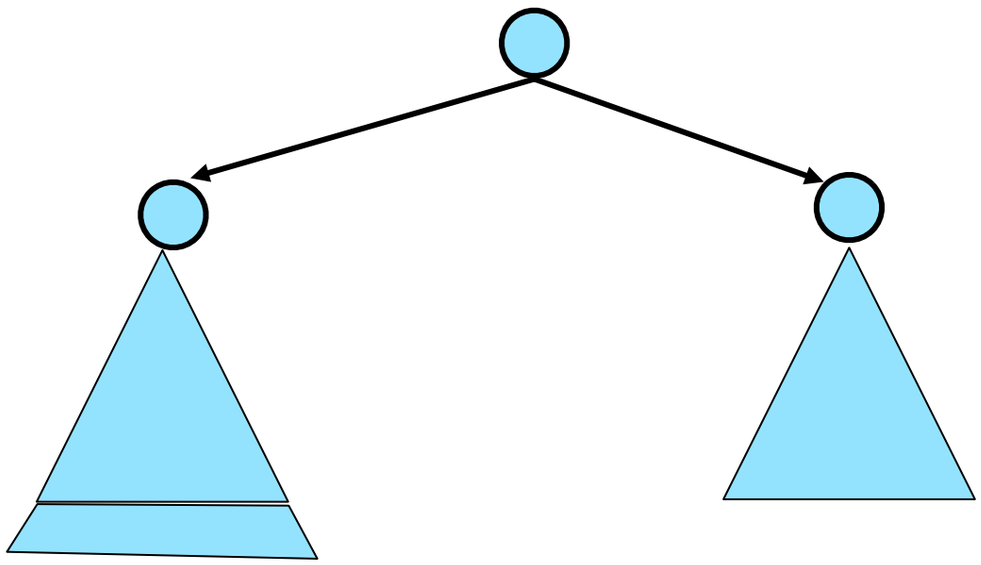
**Riassumendo i casi sì sono:**



il sinistro è completo e il destro quasi completo e h1= h2



il sinistro è quasi completo e il destro completo e h1= h2+1

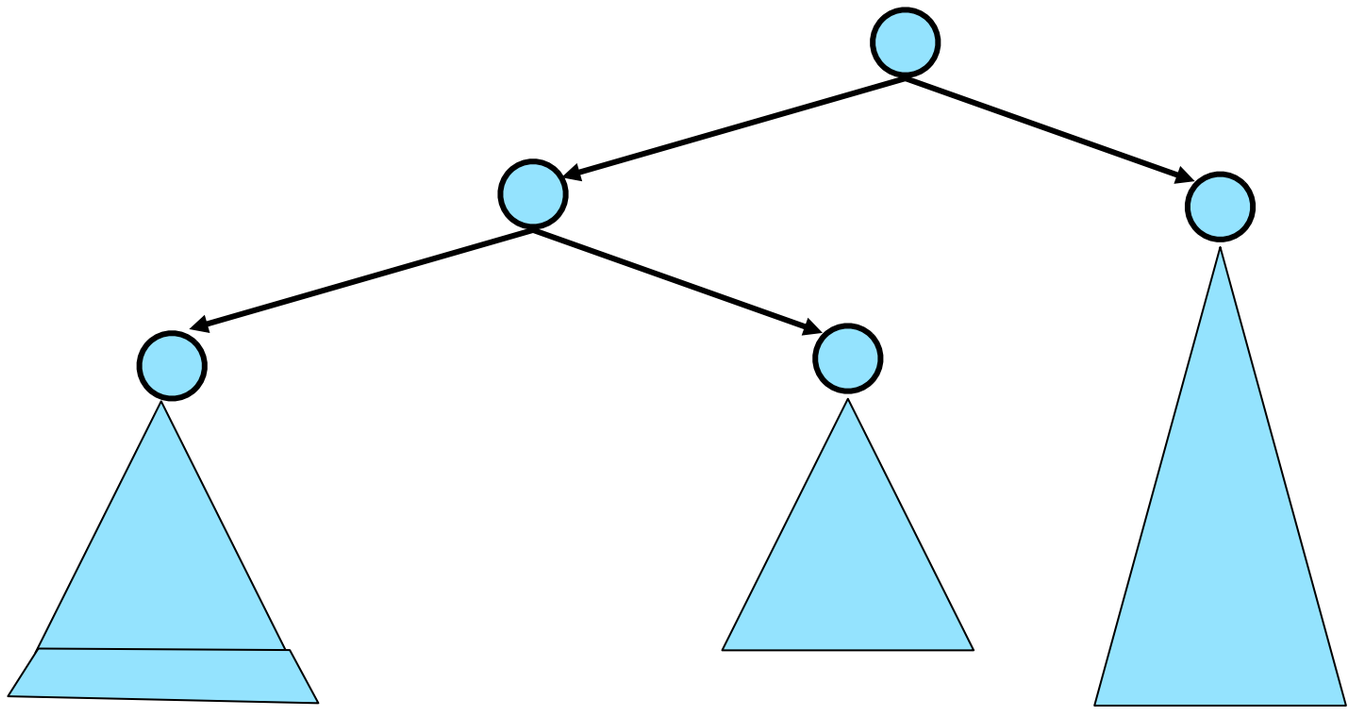


i due sottoalberi sono completi e hanno altezze h1=h2, e tutto l'albero è completo

oppure se h1= h2+1 e l'albero è quasi completo

# Esercizio 3

**Dobbiamo distinguere tra il caso in cui il sotto albero sinistro è quasi completo da quello in cui è completo**



**i due sottoalberi del sotto albero sinistro sono completi e hanno altezze h1=h2 +1,  e il sotto albero destro ha altezza h1**

**Per distinguere i due casi in cui le altezze devono essere uguali, ma in un caso si ha l'albero completo e nell'altro no, si usa una variabile booleana che viene restituita con l'altezza**

# Esercizio 3

**Si costruisca un algoritmo ricorsivo che verifica se un albero binario t, rappresentato con strutture a puntatori, è quasi completo.**

**VerQCompl(T)**

**input: il puntatore, T, alla radice di un albero binario**

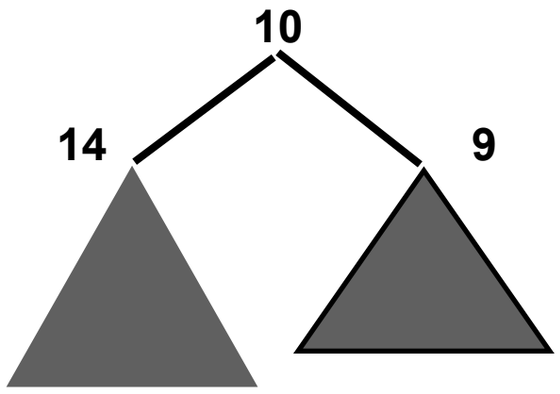
**output: una coppia (x,h) in cui h è l'altezza dell'albero e x=q se l'albero è quasi completo,mentre x=c se l'albero è completo, x=c e h=-2 altrimenti.**

```
if (T==NIL) return (c,-1)
(x,h1)=VerCompl(T.left)
(y,h2)=VerCompl(T.right)
if (h1=-2) or ( h2=-2) then return (c,-2)
elseif (x=c and y=c and h1=h2 then return (c,h1+1)
elseif (x=c and y=q and h1=h2) or
        (x=c and y=c and h1=h2+1)) then return (q,h1+1)
elseif (x=q and y=c and h1=h2+1)  then return (q,h1+1)
else return (c,-2)
```

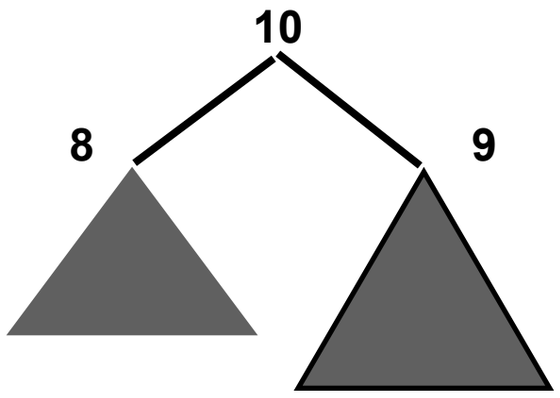
**É una visita in postordine quindi VerCompl ha tempo di esecuzione asintotico in Θ(n), nel caso peggiore, se n è il numero dei nodi dell'albero binario**

# Numero di nodi con una chiave minore di un dato valore

**Si definisca un algoritmo ricorsivo che, dato in input il puntatore alla radice di un albero binario e un intero k, dà in output il numero di nodi di un albero binario T di chiave minore a k**




**Se k = 12 allora la radice deve essere contata**




**Se k = 9 allora la radice non deve essere contata**

**Si tratta di una visita in post order, come quella per il calcolo del numero dei nodi, solo che la radice non produce sempre un incremento**

# Numero di nodi con una chiave minore di un dato valore

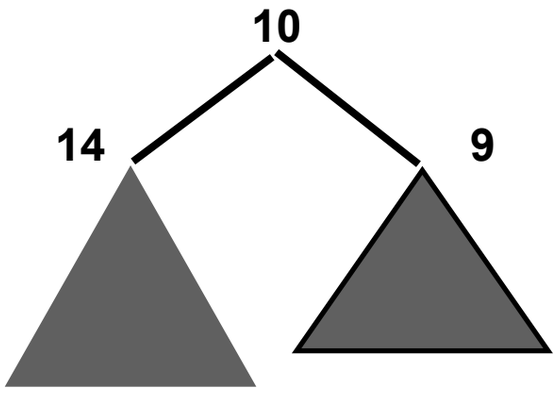
**nMinori(T, k)**
**input: un puntatore,T , alla radice di un albero binario e un intero k**
**output: il numero di elementi il cui campo intero è più piccolo di k**

```
if (T == NILL) return 0;
if (T.key < k)
return nMinori(T.left,k)+ nMinori(T.right,k) +1;
else return nMinori(T.left,k)+ nMinori(T.right,k);
```

**Tempo di esecuzione della visita Θ(n) in tutti i casi, se n è il numero dei nodi dell'albero binario.**

# Cercare una chiave in un albero binario

**Si definisca un algoritmo ricorsivo che, dato in input un albero binario T e una chiave k, dà in output il puntatore a un nodo di chiave k se presente, NIL altrimenti.**




**Se la chiave cercata non è 10 va cercata nei sottoalberi**

**Si tratta di una visita in preorder, infatti si prosegue sui sotto alberi solo se la radice non ha una chiave uguale a quella cercata**

# Cercare una chiave in un albero binario

**Si definisca un algoritmo ricorsivo che, dato in input un albero binario T e una chiave k, dà in output il puntatore a un nodo di chiave k se presente, NIL altrimenti.**

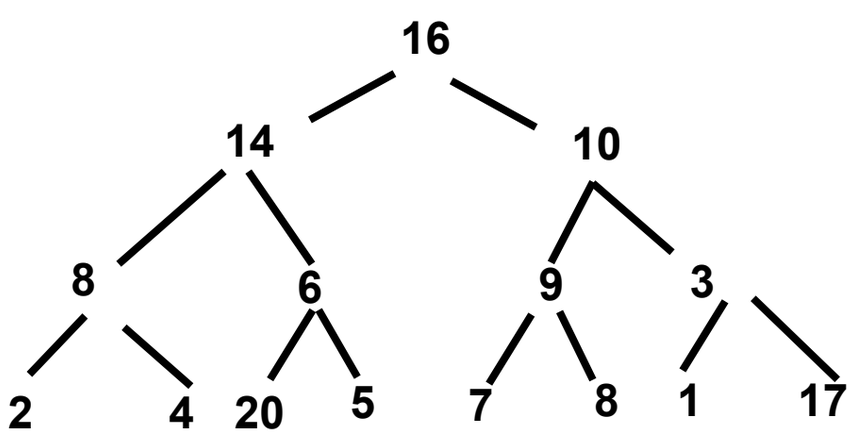
**cerca(T, x)**
**input: un puntatore, T, alla radice di un albero binario e x un intero**
**output:il puntatore al primo nodo dell'albero di chiave x se lo trova, NIL altrimenti**

```
if T== NIL return NIL
if (T.key == x) return T
temp = cerca(T.left,x)
if (temp == NIL) return cerca(T.right,x)
return temp
```

**Tempo di esecuzione della visita Θ(n) nel caso peggiore e Θ(1) nel migliore.**

# Nodi su un cammino

**Si definisca un algoritmo ricorsivo che, dato un albero binario T, le cui chiavi non si ripetono, e due chiavi x e y, dà in output 1 se c'è un cammino padri-figli che porta da un nodo di chiave x a uno di chiave y, 0 altrimenti.**




**Per esempio 10 e 8 sono su un cammino padri-figli, mentre 6 e 10 no**

**L'idea è cercare x e poi cercare y nel sotto albero radicato in x.**

# Nodi su un cammino

**Si definisca un algoritmo ricorsivo che dà in output 1 se c'è un cammino padri-figli che porta da un nodo di chiave x a uno di chiave y, 0 altrimenti.**

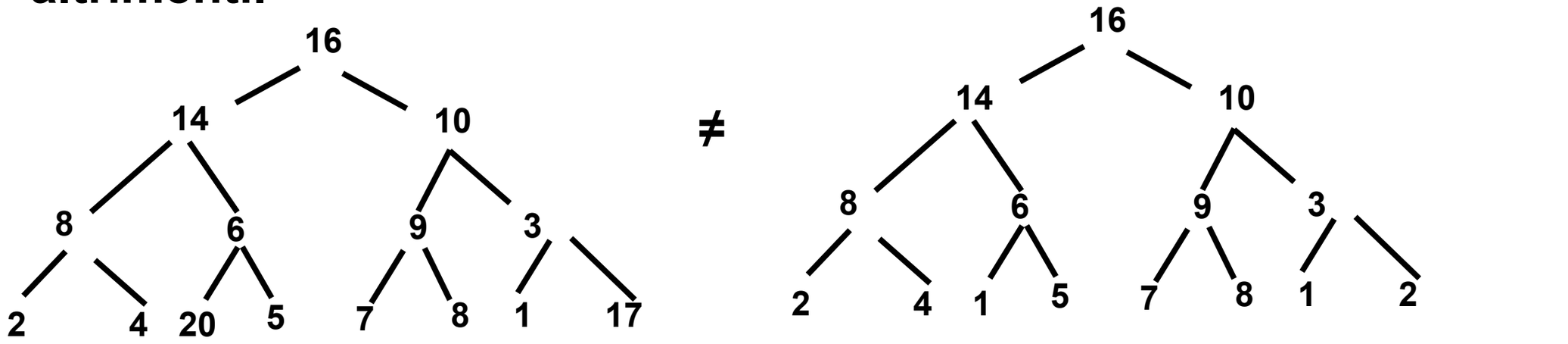
**Cammino(T, x, y)**
**input: un puntatore, T, alla radice di un albero binario e due interi x e y**
**prec: le chiavi in T non si ripetono**
**output:1 se c'è un cammino padri-figli che porta dal nodo di chiave x a quello di chiave y, 0 altrimenti.**

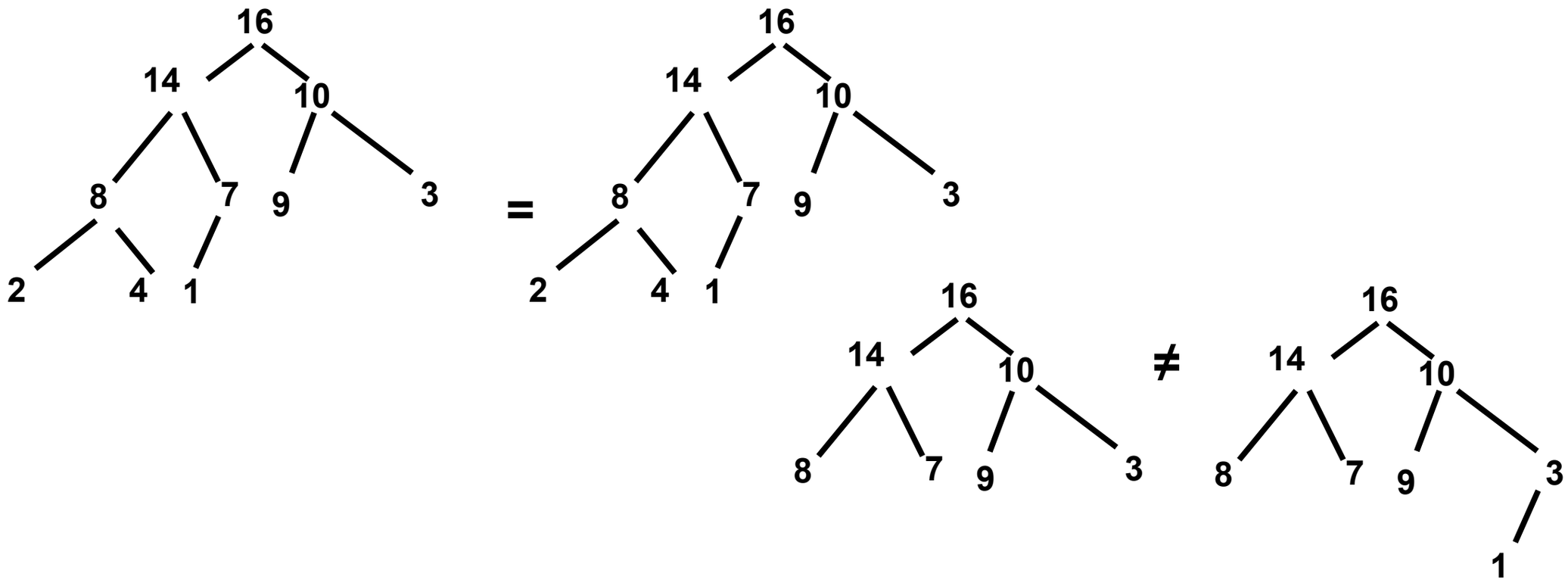
```
if (cerca(cerca(T,x),y)) ≠ NIL return 1
  else return 0
```

**Tempo di esecuzione Θ(n) nel caso peggiore, dove n è il numero dei nodi, per esempio se x non è presente. Se si trova x poi la visita si riduce al sotto albero radicato in x. Θ(1) nel caso migliore, per esempio x è la radice e y uno dei suoi figli.**

# Uguaglianza tra due alberi

**Si definisca un algoritmo ricorsivo che, dati in input i puntatori alle radici di due alberi binari T e U, dà in output 1 se T=U, 0 altrimenti.**



# Uguaglianza tra due alberi

**Si definisca un algoritmo ricorsivo che, dati in input i puntatori alle radici di due alberi binari T e U, dà in output 1 se T=U, 0 altrimenti.**

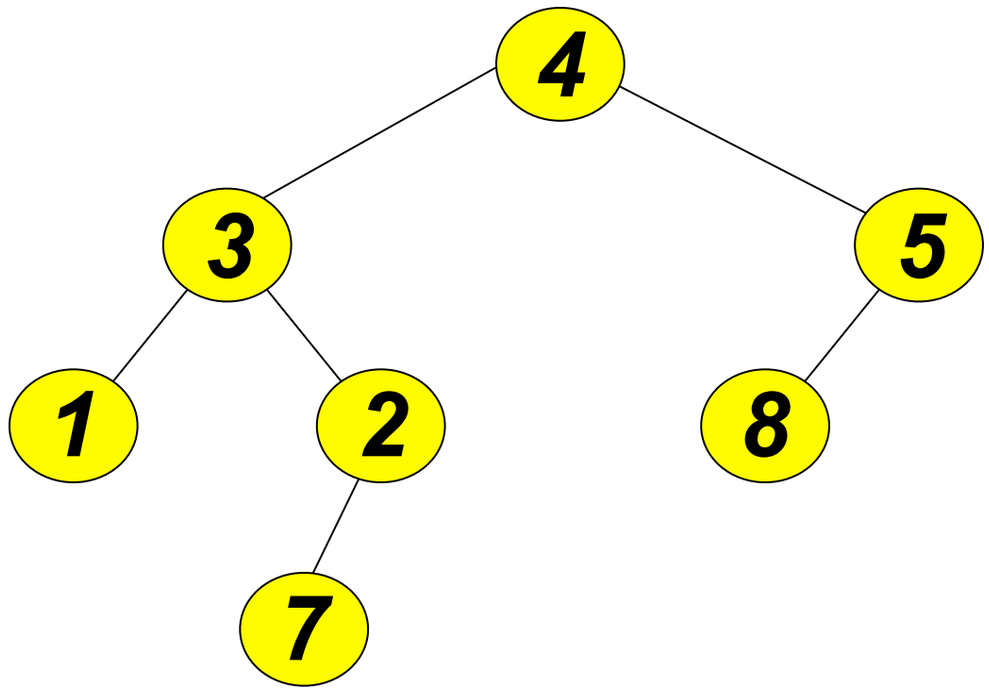
```
Uguali(T, U)
input: due puntatori, T e U , alla radice di due albero binari
output:1 se T e U sono uguali, 0 altrimenti.
if (T == NIL and  U == NIL) return 1
se sono entrambi vuoti sono uguali
else if (T == NIL  and U ≠ NIL ) or (T≠ NIL and U == NIL) return 0
if (T.key == U.key)
                return Uguali(T.left,U.left) and Uguali(T.right,U.right)
 else return 0
```

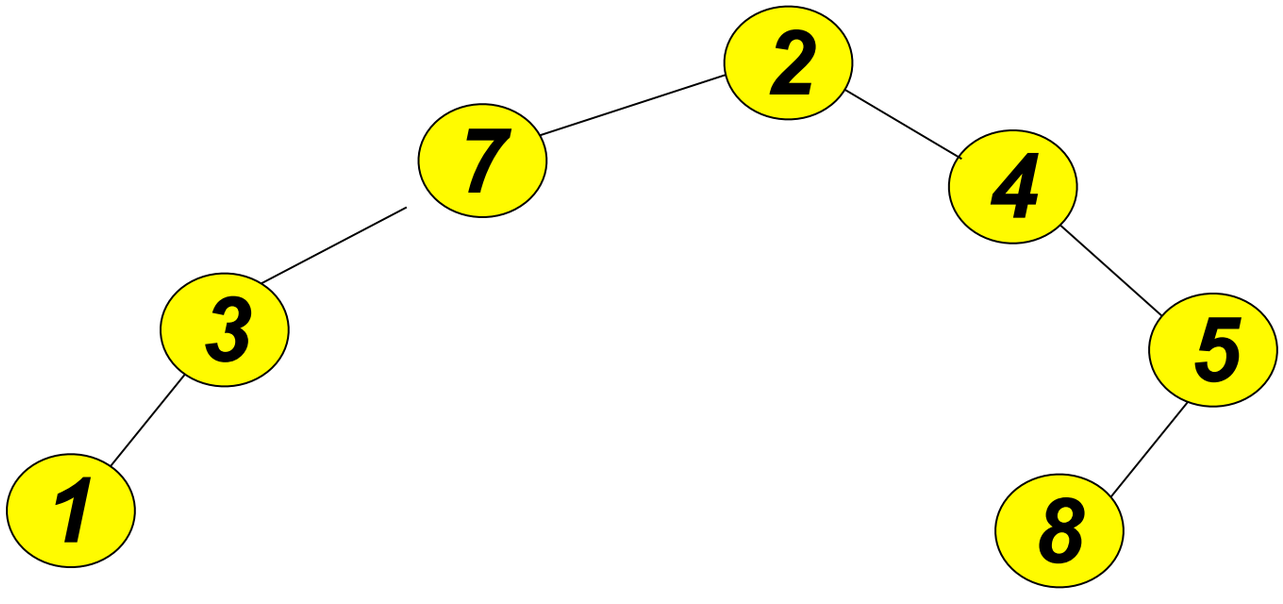
**Detto n il numero dei nodi dell'albero più piccolo, il tempo di esecuzione è Θ(n) nel caso peggiore, se i due alberi sono uguali, oppure il primo è "prefisso" del secondo. Θ(1) nel caso migliore, uno è vuoto e l'altro no, oppure le chiavi delle radici sono diverse.**

# Alberi da visite

**visita inordine: 1 3 7 2 4 8 5**

**visita preordine: 4 3 1 2 7 5 8**



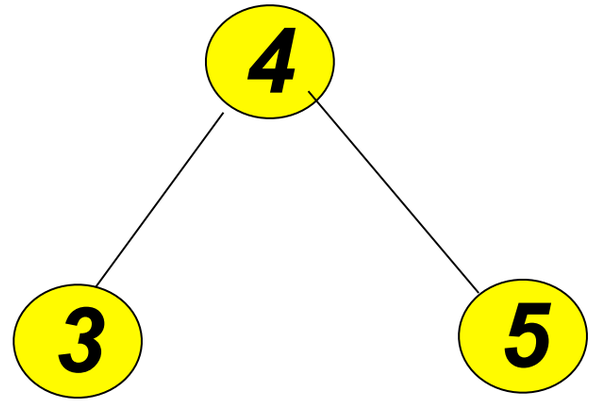

**visita inordine: 1 3 7 2 4 8 5**

**visita preordine: 2 7 3 1 4 5 8**

# Alberi da visite

**visita inordine: 1 3 7 2 4 8 5**

**visita preordine: 4 3 1 2 7 5 8**




**Nella visita in preordine il primo è la radice**

**Nella visita inordine il sottoalbero sinistro è visitato prima della radice, quindi**
**1 3 7 2 sono nodi del sottoalbero sinistro e 8 5 del destro.**

**Applicando ricorsivamente il ragionamento ai due sottoalberi, si ottiene che 3 è la radice del sottoalbero sinistro e 5 del destro.**

# Alberi da visite

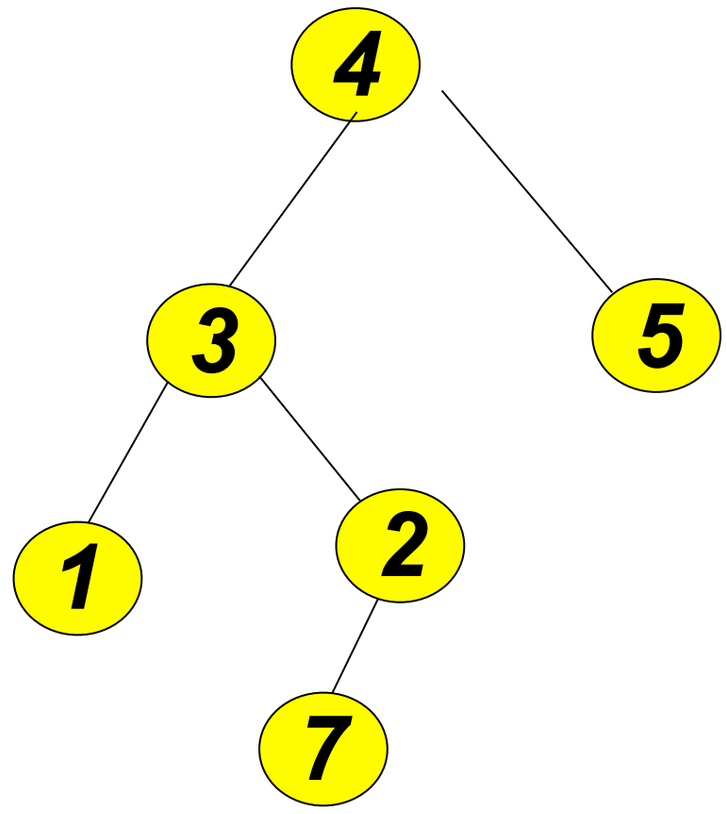
**visita inordine: 1 3 7 2 4 8 5**

**visita preordine: 4 3 1 2 7 5 8**

**1 3 7 2 sono i nodi del sottoalbero sinistro e 8 5 quelli del destro.**

**Dalla visita inordine del sottoalbero sinistro ricaviamo che 1 è l'unico nodo nel sottoalbero sinistro di 3 e 7 2 in quello destro.**

**Poichè nella visita preordine 2 è visitato prima di 7, 2 è la radice e 7 sta nel sottoalbero sinistro perchè precede 2 nella visita inordine.**

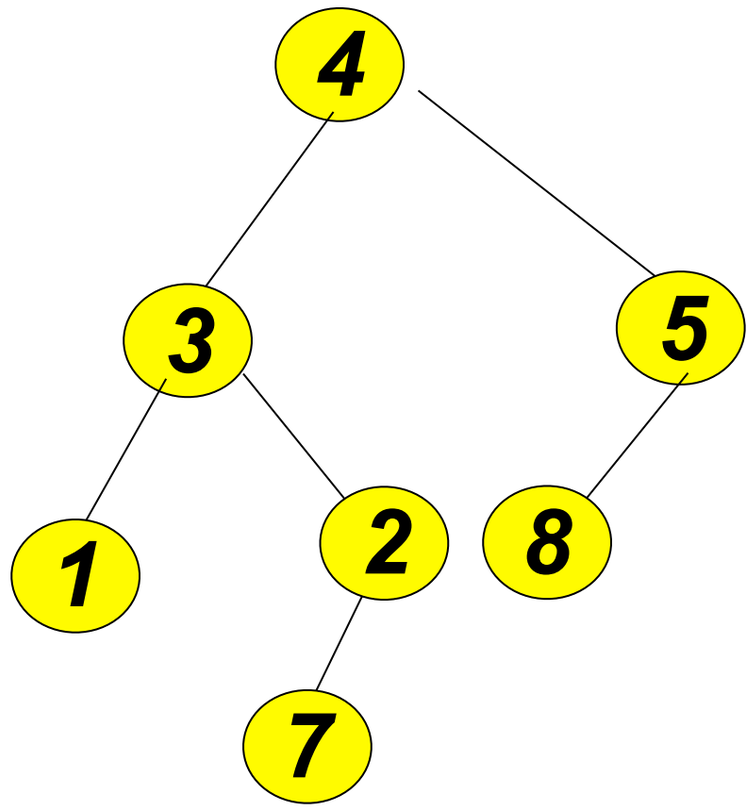


# Alberi da visite

**visita inordine: 1 3 7 2 4 8 5**

**visita preordine: 4 3 1 2 7 5 8**

**1 3 7 2 sono i nodi del sottoalbero sinistro e 8 5 quelli del destro.**




**Poichè nella visita inordine 8 è visitato prima di 5, 8 sta nel sottoalbero sinistro**

# Alberi da visite

**CostrAB(inordine, preordine)**

**Input: due arrays inordine e preordine con le visite inorder e preorder rispettivamente,**

**prec:inordine e preordine sono non  vuoti, di elementi distinti**

**output: il puntatore alla radice dell'albero binario T, la cui visita inorder è inordine e quella in preorder è preordine**

**in_lo = 0; in_hi = inordine.len - 1; pre_lo= 0; pre_hi=in_hi**

**return CostrAB_aus(inordine, preordine,in_lo,in_hi,pre_lo,pre_hi)**

**Si utilizza una funzione di ricerca in un vettore**

**cerca(inordine,in_lo,in_hi,x)**

**Input: un vettore inordine, due indici in_lo e in_hi che individuano l'intervallo di ricerca in inordine e la chiave x da cercare.**

**prec:inordine è non  vuoto, di elementi distinti e lo≤hi**

**postc: restituisce la posizione in in di x**

# Alberi da visite

**CostrAB_aus(inordine, preordine,in_lo,in_hi,pre_lo,pre_hi)**

**Input: due sequenze inordine e preordine con le visite inorder e preorder rispettivamente,**

**in_lo e in_hi sono gli indici che individuano la porzione di inordine su cui si costruisce l'albero, mentre pre_lo e pre_hi sono gli indici che individuano la porzione di preordine su cui si costruisce l'albero**

**prec:inordine e preordine contengono elementi distinti**

**output: dà in output il puntatore alla radice dell' albero binario T, la cui visita inorder è inordine e quella in preorder è preordine**

```
if lo > hi  then return NIL
crea un nodo T, con T.key = preordine[pre_lo] e campi puntatori a NIL
if lo = hi  then return T
 j = cerca(inordine,in_lo,in_hi,T.key)
```

**j è l'indice della radice in inordine, il sottoalbero sinistro contiene i nodi inordine[in_lo…j-1] e il destro i nodi inordine[j+1 ... in_hi]. La porzione di preordine che contiene i nodi del sotto albero sinistro è compresa tra l'indice pre_lo+1 e quello ottenuto sommando il numero di elementi nel sotto albero sinistro, j-in_lo, a pre_lo.**

**Invece i nodi nel sotto albero destro in preordine sono quelli a seguire, quindi dall'indice pre_lo+1 + (j-in_lo) e pre_hi**

```
T.left = CostrAB_aus(inordine, preordine,in_lo,j-1,pre_lo+1,pre_lo + (j - in_lo))
T.right = CostrAB_aus(inordine, preordine,j+1,in_hi,pre_lo + (j - in_lo)+1,pre_hi)
return T
```

# Alberi da visite

**Relazioni di ricorrenza per CostrAB_aus.**

**$T(n) = T(0) + T(n-1) + cn$ che si semplifica con**
**$T(n) = T(n-1) + cn$**
**Sappiamo che $T(n) = O(n^2)$.**
**Verifichiamo che $T(n) \leq kn^2$, per una costante $k>0$, a partire da un certo n in poi. Usiamo l'induzione.**
**$T(n) = T(n-1) + cn \leq$ (per l'ipotesi induttiva)**
**$k(n-1)^2 + cn =$**
**$k(n^2 + 1 - 2n) + cn =$**
**$kn^2 + k - 2kn + cn$**
**Imponendo la disuguaglianza:**
**$kn^2 + k - 2kn + cn \leq kn^2$**
**$\Leftrightarrow$ $k - 2kn + cn \leq 0$**

**$\Leftrightarrow$ $2kn - k \geq cn$**

**Questa disuguaglianza è vera per $k = c$ e $n \geq 1$**

# Alberi da visite

**Relazioni di ricorrenza per CostrAB_aus.**

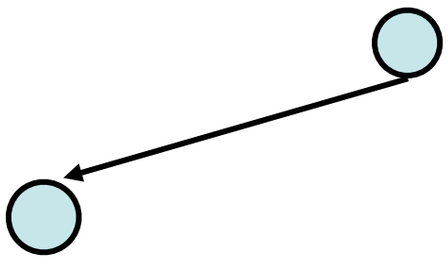
**T(n) = T(n/2) + T(n/2) + cn/2 che si semplifica con**
**T(n) = 2T(n/2) + cn**
**Sappiamo che T(n) = Θ(n lg n).**

# Numero di nodi contenti

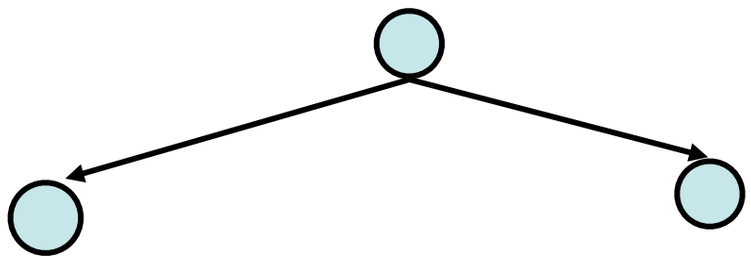
**Si definisca un algoritmo ricorsivo che dà in output il numero di nodi contenti nel'albero di input, se non è vuota e  -1 altrimenti. Un nodo e' contento se nell'albero in esso radicato ci sono piu' nodi figli sinistri che destri.**

# nodi contenti

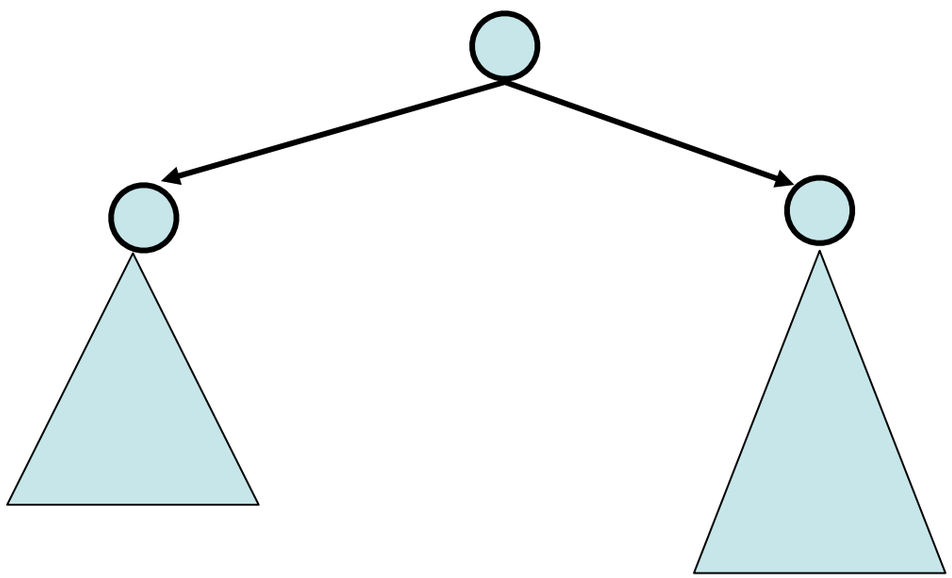
**non è contento**



**è contento ed è l'unico contento, dare 1**



**nessuno è contento, dare 0**



**se si hanno n1 nodi contenti nel sotto albero sinistro e n2 in quello destro e se il numero dei nodi nel sotto albero sinistro è minore di quello nel sotto albero destro allora dare n1+n2+1, altrimenti  n1+n2**

# Numero di nodi contenti, pseudocodicce

**Si definisca un algoritmo ricorsivo che dà in output il numero di nodi contenti nel'albero di input, se non è vuota e -1 altrimenti. Un nodo e' contento se nell'albero in esso radicato ci sono piu' nodi figli sinistri che destri.**

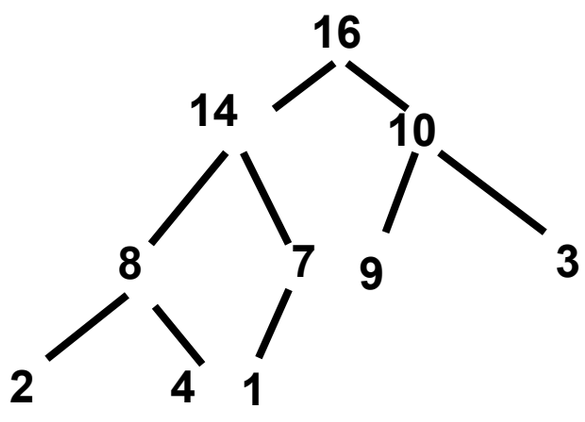
**Il tempo di esecuzione è quello di una visita in postordine, quindi Θ(n), dove n è il numero dei nodi di T.**

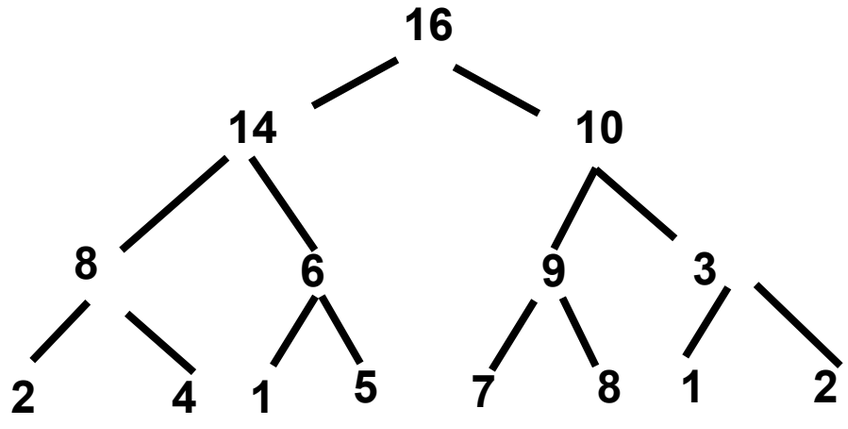
```
contenti(T)
input: il puntatore alla radice di un albero binario
output: il numero dei nodi contenti in T
(n,contenti) = contAux(T);
return contenti}
```

```
contAux(T)
input: il puntatore alla radice di un albero binario e una variabile intera n
output: la coppia con il numero dei nodi e il numero dei nodi contenti in T
if T è nil  return (0,0)
if T è una foglia return (1,0)
if (T.left ≠ nil) then (n1,contenti)  =  contAux(T.left)
if (T.right ≠ nil) then (n2,contenti) = contAux(T.right)
if  (n1 - n2  > 0)  then return (n1+n2+1,contenti+1) else
                         return (n1+n2+1,contenti)
```

# Ultima Foglia

**Dato un albero binario T, quasi completo (cioè completo fino al penultimo livello, con le foglie al più su due livelli e con le foglie sull'ultimo livello tutte a sinistra) implementato con strutture a puntatori, si implementi un algoritmo che ricevuto in input T restituisce in output il puntatore all'ultima foglia.**
**L'algoritmo deve avere una complessita' $O((\lg n)^2)$, se n è il numero degli elementi in T.**
**Si illustri l'idea algoritmica prima di passare alla stesura in pseudocodice, in cui prima di tutto si deve indicare l'output atteso delle singole funzioni utilizzate, oltre ad eventuali vincoli sull'input.**
**Esempio: nel primo albero la risposta è il link al nodo 1, nel secondo al nodo 2 più a destra.**



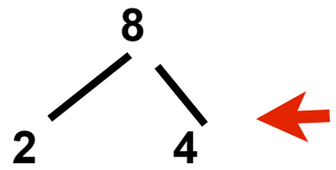



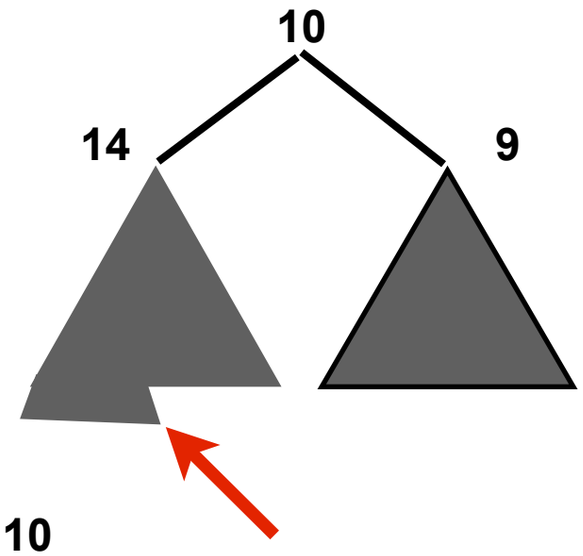
# Analisi problema

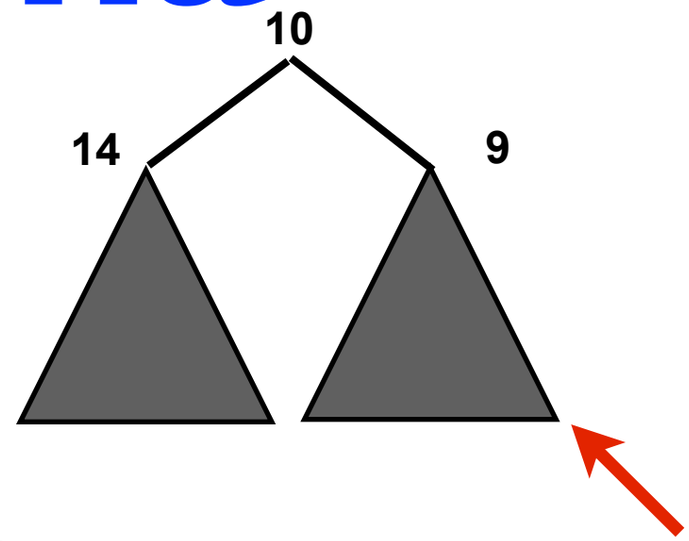
**Casi base:**

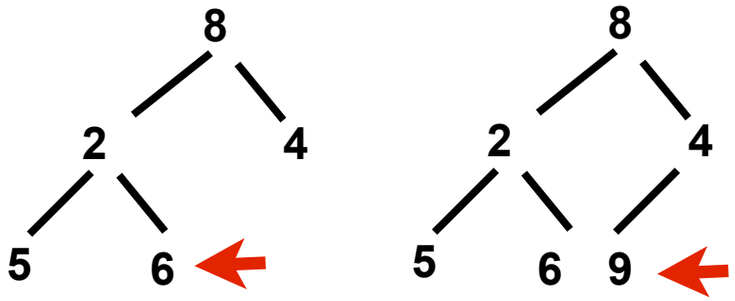






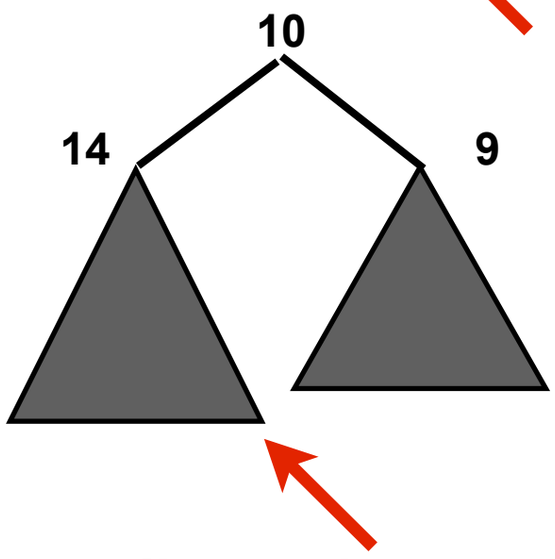



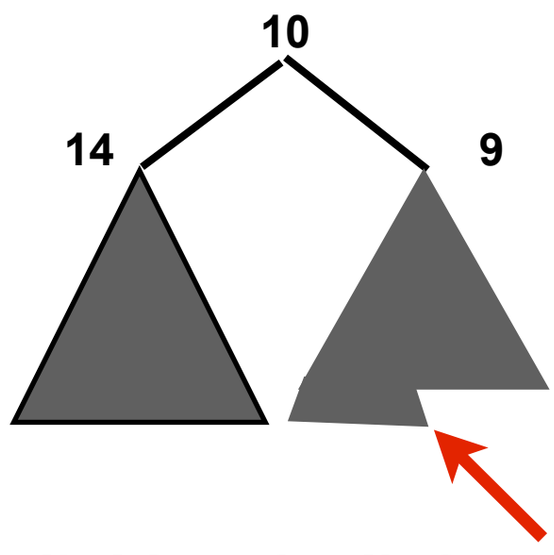





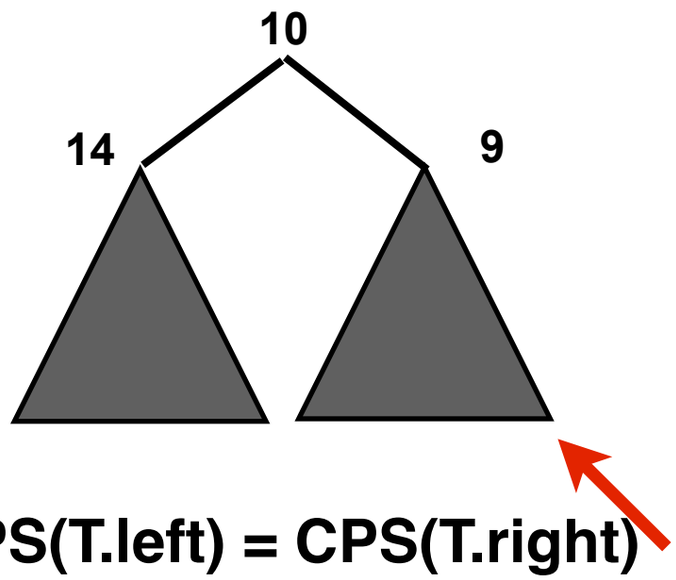




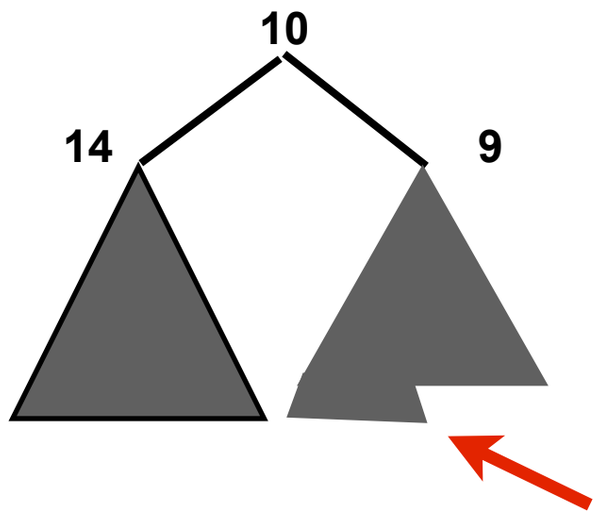




**Come posso individuare il sotto albero che contiene l'ultima foglia?**
**Osserviamo che se la lunghezza del cammino più a sinistra nel sotto albero sinistro e destro sono uguali allora l'ultima foglia è nel sotto albero destro, altrimenti nel sinistro.**
**Sia CPS la funzione che restituisce il numero di nodi attraversati lungo un cammino che parte dalla radice e scende lungo legami padre-figlio sinistro fino a che il nodo è privo di figlio sinistro.**

# Verso la soluzione

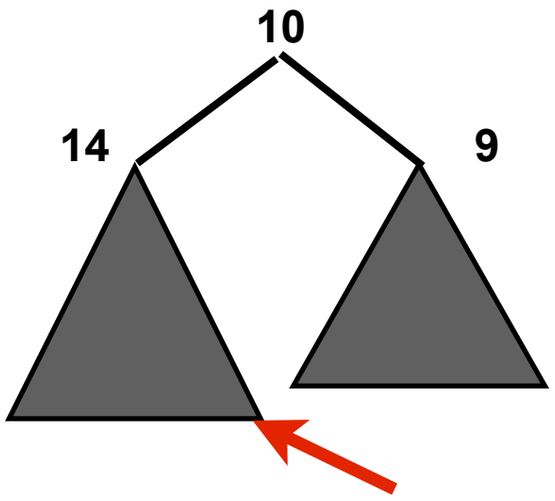



CPS(T.left) = CPS(T.right)

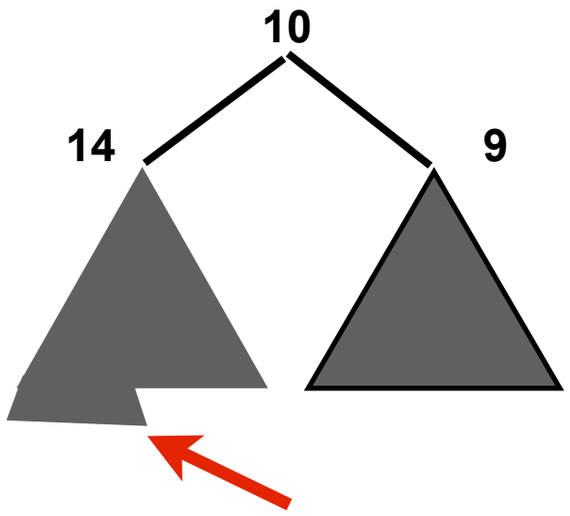



CPS(T.left) = CPS(T.right)

**se CPS(T.left) = CPS(T.right)**
**dirigo la ricerca nel sottoalbero destro**




CPS(T.left) = CPS(T.right) + 1




CPS(T.left) = CPS(T.right) + 1

**se CPS(T.left) = CPS(T.right) + 1**
**dirigo la ricerca nel sottoalbero sinistro**

**Tempo di esecuzione di CSP Θ(h) = Θ(lg n)**

# Pseudocodice

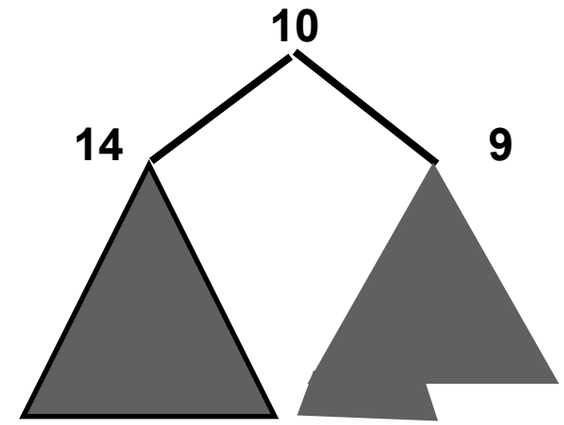
**UltimaFoglia(T)**

Input: un albero binario T

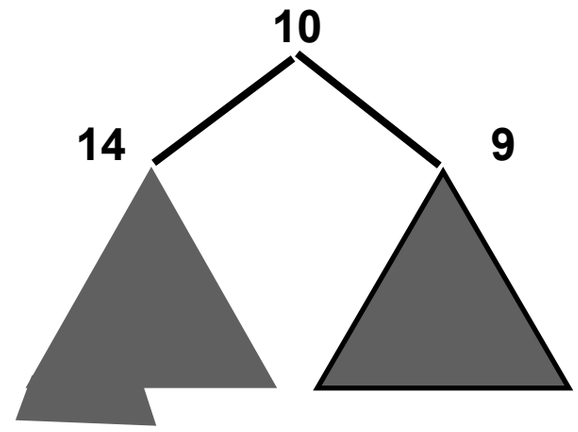
precond: T è quasi completo

output: il puntatore all'ultima foglia

```
if (T == NIL o è una foglia)  then return   T
m = CPS(T.left)
m' = CPS(T.right)
if (m == m' )  then return UltimaFoglia(T.right)
if (m == m' + 1)  then return UltimaFoglia(T.left)
```




m = m'




m = m' +1

**CPS(T)**

Input: un albero binario T

output: il numero di nodi attraversati lungo un cammino che parte dalla radice e scende lungo legami padre-figlio sinistro fino a che il nodo è privo di figlio sinistro.

%Se T è Nil dà 0 se T è un solo nodo dà 1

**Tempo di esecuzione $T(h) = T(h-1) + \Theta(h)$ in tutti i casi.**

**Quindi $\Theta(h^2)$ e $h = \lg n$.**