In questa lezione

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

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

Rappresentazione in memoria alberi binari

Dato un albero binario t.

Nella rappresentazione in memoria assumiamo che

ogni nodo di t abbia

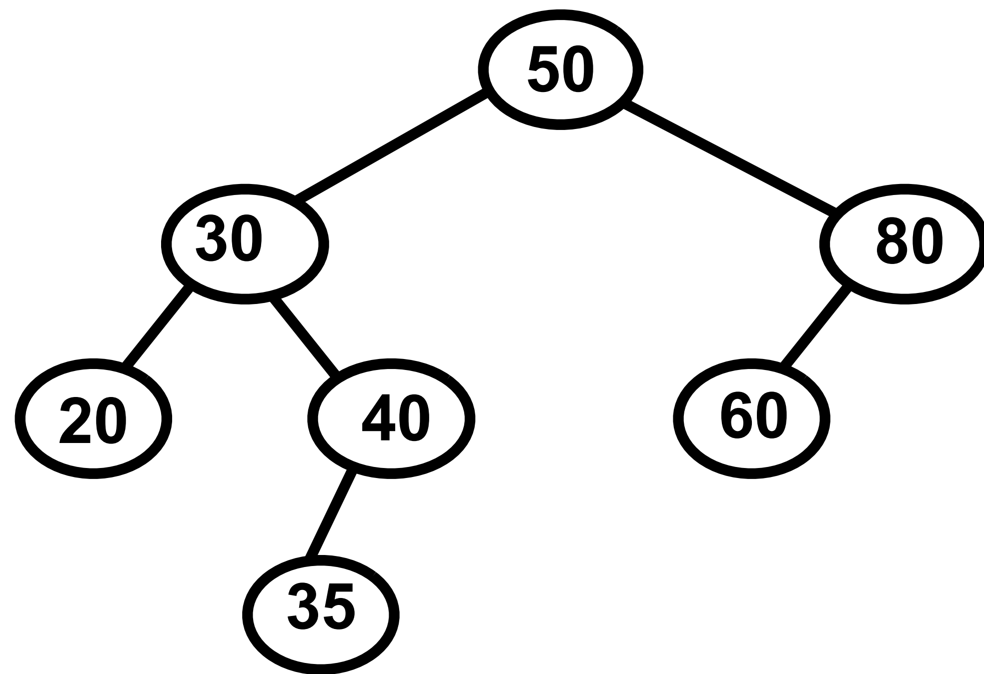
un campo key, che contiene la chiave del dato

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



nil
50

p
key
left
30
right

80
nil

20
nil nil

40
nil

60
nil nil

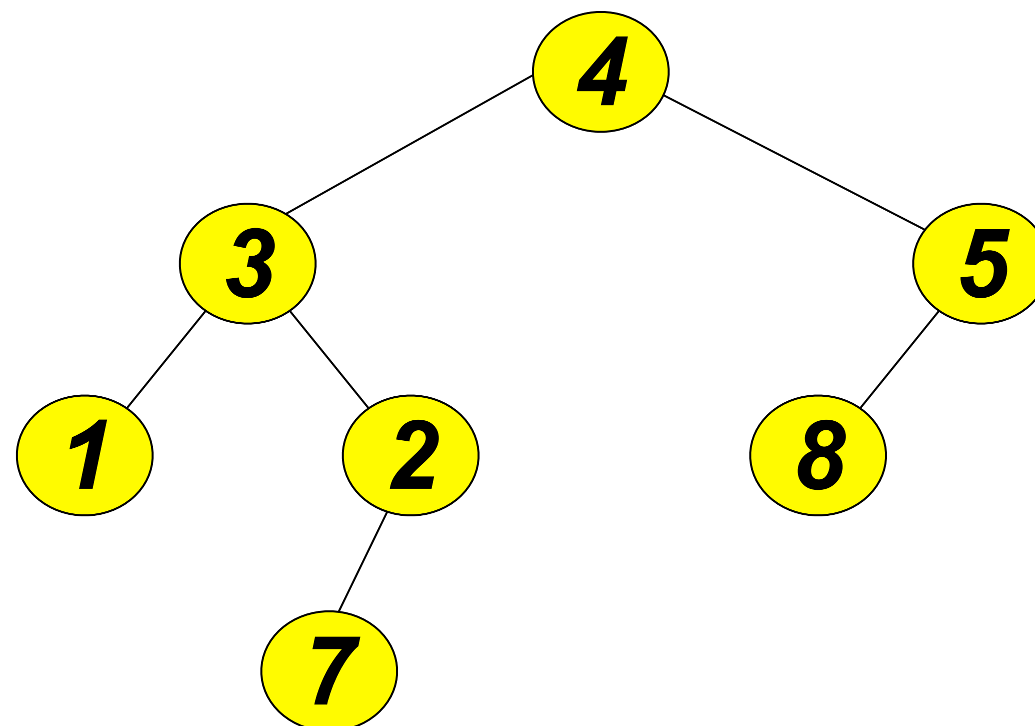
35
nil nil

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

Quindi nella visita inordine la radice è visitata dopo la visita del suo sotto albero sinistro e prima del sotto albero destro, per questo si chiama in ordine

Input:



Output: 1372485

Visita inordine di un albero binario

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

Inordine(nodo di chiave 4)

Inordine(nodo di chiave3)

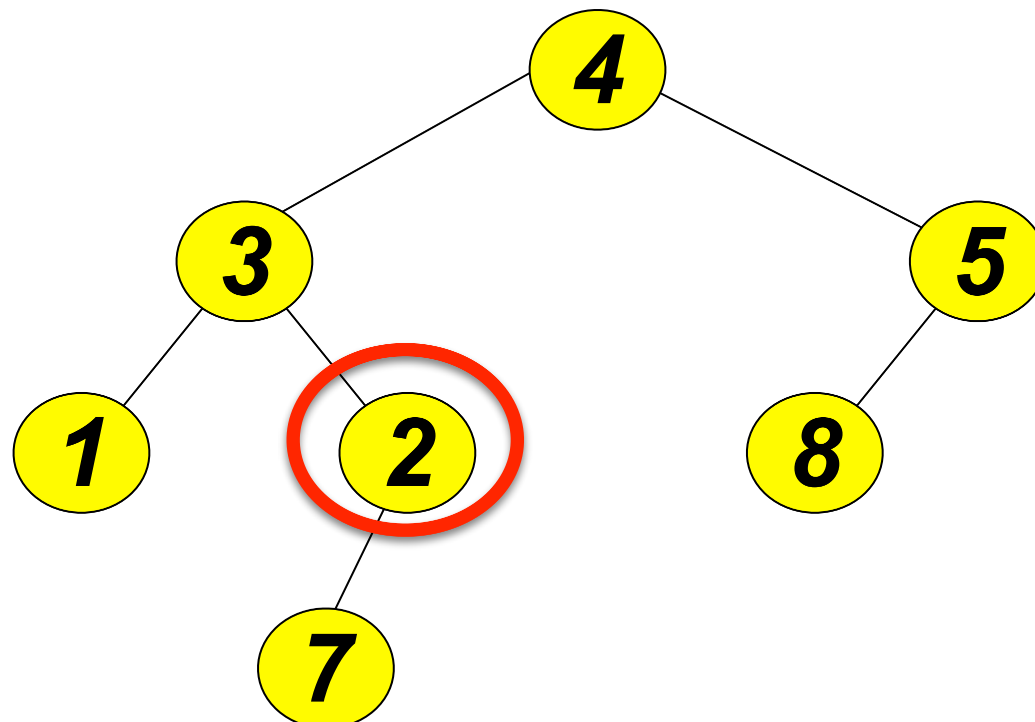
Inordine(nodo di chiave1)

Inordine(nil)

print key(1)

print key(3)

Input:



Output: 1 3

Visita inordine dopo la conclusione della chiamata sul nodo di chiave 1

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

Inordine(nodo di chiave 4)

Inordine(nodo di chiave3)

Inordine(nodo di chiave2)

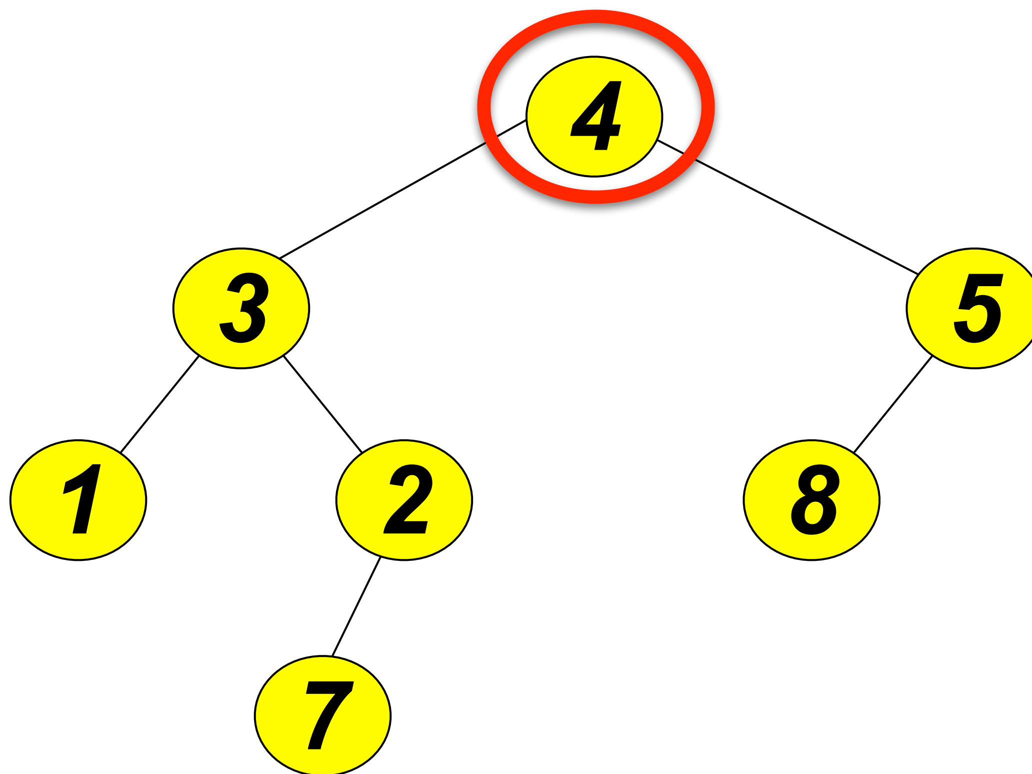
Inordine(nodo di chiave7)

print key(7)

print key(2)

print key(4)

Input:



Output: 1 3 7 2 4

Visita inordine rientro nella prima chiamata

Inordine(nodo di chiave 4)

Inordine(nodo di chiave 5)

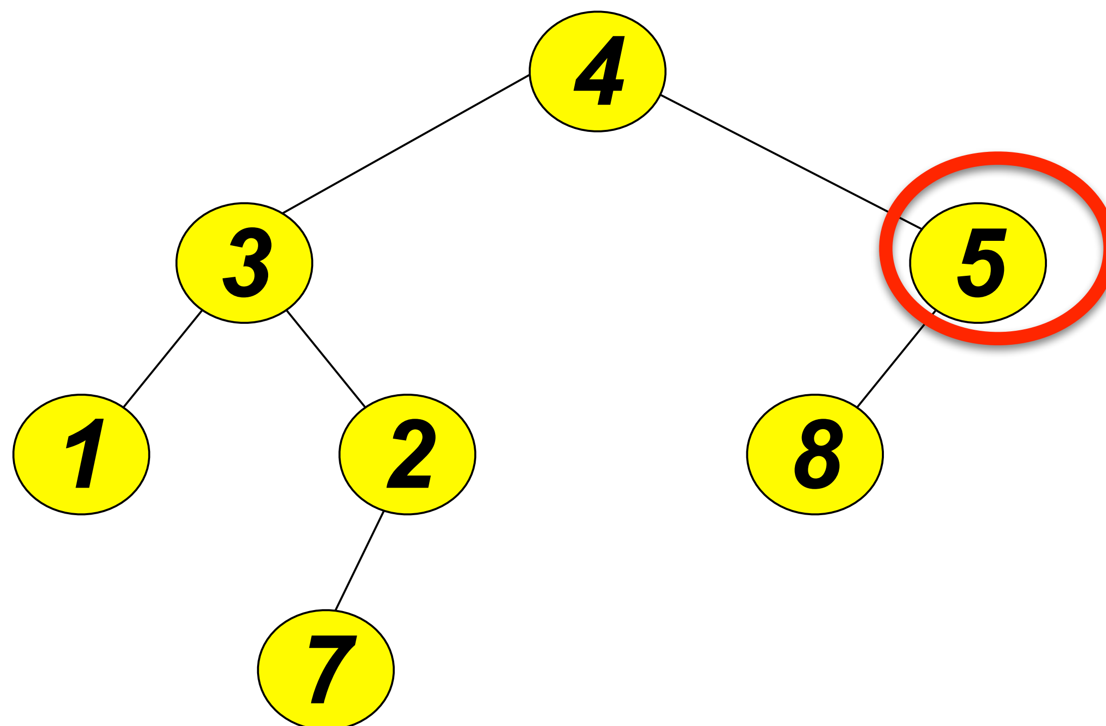
Inordinenodo di chiave 8)

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

print key(8)

print key(5)

Input:



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])
```

Prevediamo che $T(n) = O(n)$, cioè che esistono due costanti positive k e n_0 tali che $T(n) \leq kn$, per ogni $n \geq n_0$.
Per ipotesi induttiva $T(m) = O(m)$ per ogni $m < n$

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

si vuole che $kn - k + d \leq kn$ e 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 $n \geq n_0=1$.

Quindi $T(n) = O(n)$

Complessità visita inordine

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

$T(n) = \Omega(n)$, perchè tutti i nodi sono visitati.

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

Postorder

Modificando l'ordine di visita della radice rispetto ai suoi sottoalberi si ottengono visite diverse.
Nella visita in postordine la radice è visitata dopo i suoi sotto alberi

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

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

Preorder

Modificando l'ordine di visita della radice rispetto ai suoi sottoalberi si ottengono visite diverse.
Nella visita in preordine la radice è visitata prima dei suoi sotto alberi

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)
```

Complessità visite

Sia la visita in preordine che quella in postordine hanno la stessa complessità della inordine:

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

Infatti la relazione di ricorrenza è la stessa!

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. Quindi il tempo di esecuzione dell'algoritmo per il calcolo del numero dei nodi, n, è $\theta(n)$.

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
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. Quindi il tempo di esecuzione dell'algoritmo per il calcolo dell'altezza, in un albero con n nodi, è $\theta(n)$.

Altri conteggi: altezza 2

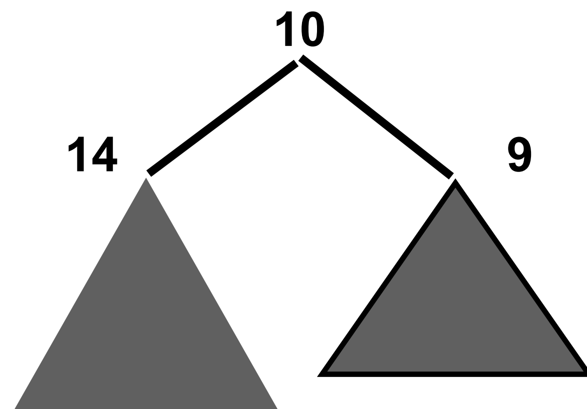
```
Altezza(x)
input: x è il puntatore alla radice di un albero binario
output: l'altezza dell'albero di radice x

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

Versione più sintetica e veloce!

Cercare una chiave in un albero binario

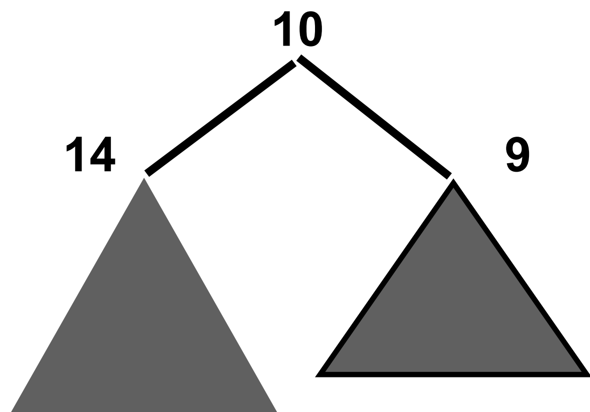
Si definisca un algoritmo ricorsivo che, dato in input un albero binario T e una chiave k, dà in output il puntatore a un nodo di chiave k se presente, NIL altrimenti.
Esempio:



Se la chiave cercata è 10 visitando la radice si dà il puntatore alla radice in output, altrimenti la chiave va cercata nei sottoalberi

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.



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.
Si cerca nel sotto albero sinistro e solo se non la si trova si cerca anche nel sotto albero destro.

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.

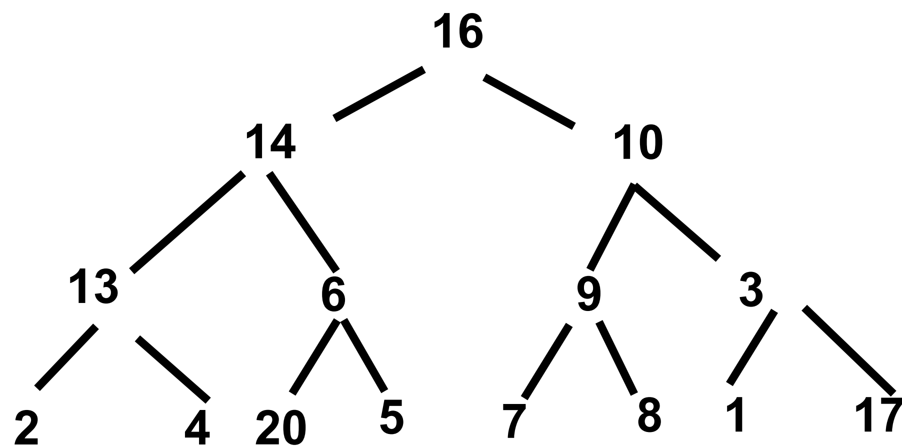
cerca(T, x)
input: un puntatore, T, alla radice di un albero binario e x un intero
output:il puntatore al primo nodo dell'albero di chiave x se lo trova, NIL altrimenti

```
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 $\Theta(n)$ nel caso peggiore e $\Theta(1)$ nel migliore.

Nodi su un cammino

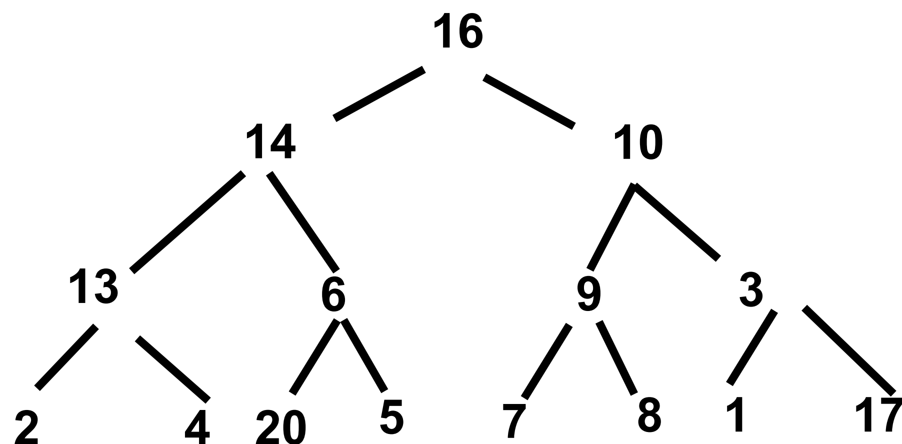
Si definisca un algoritmo ricorsivo che, dato un albero binario T, le cui chiavi non si ripetono, e due chiavi x e y, dà in output 1 se c'è un cammino padri-figli, che non ha la radice come nodo intermedio per la radice, che porta da un nodo di chiave x a uno di chiave y, 0 altrimenti.



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

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
```

Nodi su un cammino: analisi

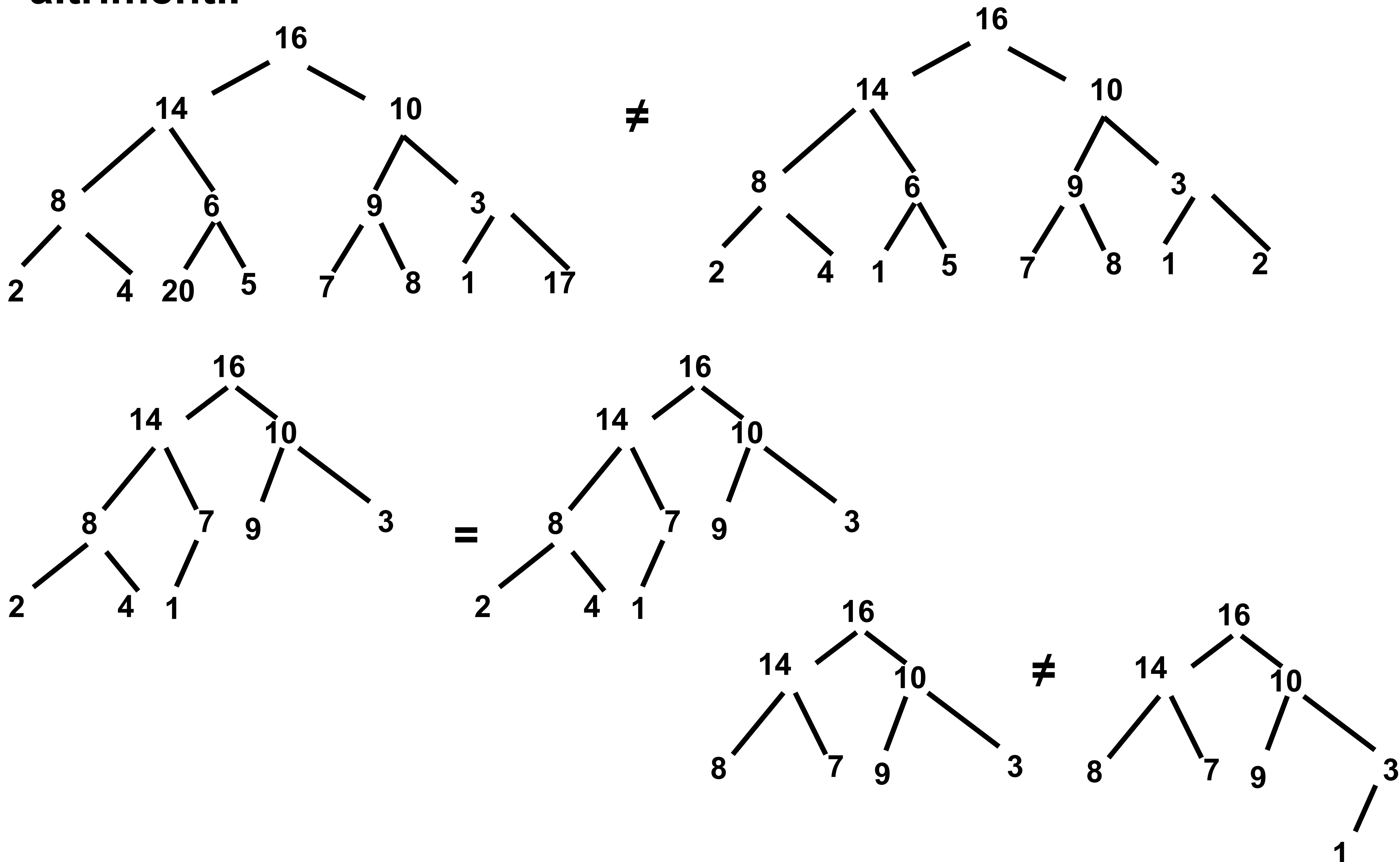
Cammino(T, x, y)
input: un puntatore, T, alla radice di un albero binario e due interi x e y
prec: le chiavi in T non si ripetono
output:1 se c'è un cammino padri-figli che porta dal nodo di chiave x a quello di chiave y, 0 altrimenti.

```
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 tratta di due visite in preordine sui due alberi, le visite dei nodi consistono nel controllare che entrambi gli alberi abbiano un nodo da visitare e che le chiavi siano uguali.

```
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
if (T == NIL  and U ≠ NIL ) or (T≠ NIL and U == NIL) return 0
if (T.key == U.key) then
                return Uguali(T.left,U.left) and Uguali(T.right,U.right)
  else return 0
```

Uguaglianza tra due alberi

Si tratta di due visite in preordine sui due alberi, le visite dei nodi consistono nel controllare che entrambi gli alberi abbiano un nodo da visitare e che le chiavi siano uguali.

```
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
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, per esempio se i due alberi sono uguali, oppure il primo è "prefisso" del secondo. Θ(1) nel caso migliore, per esempio se le chiavi delle radici sono diverse.