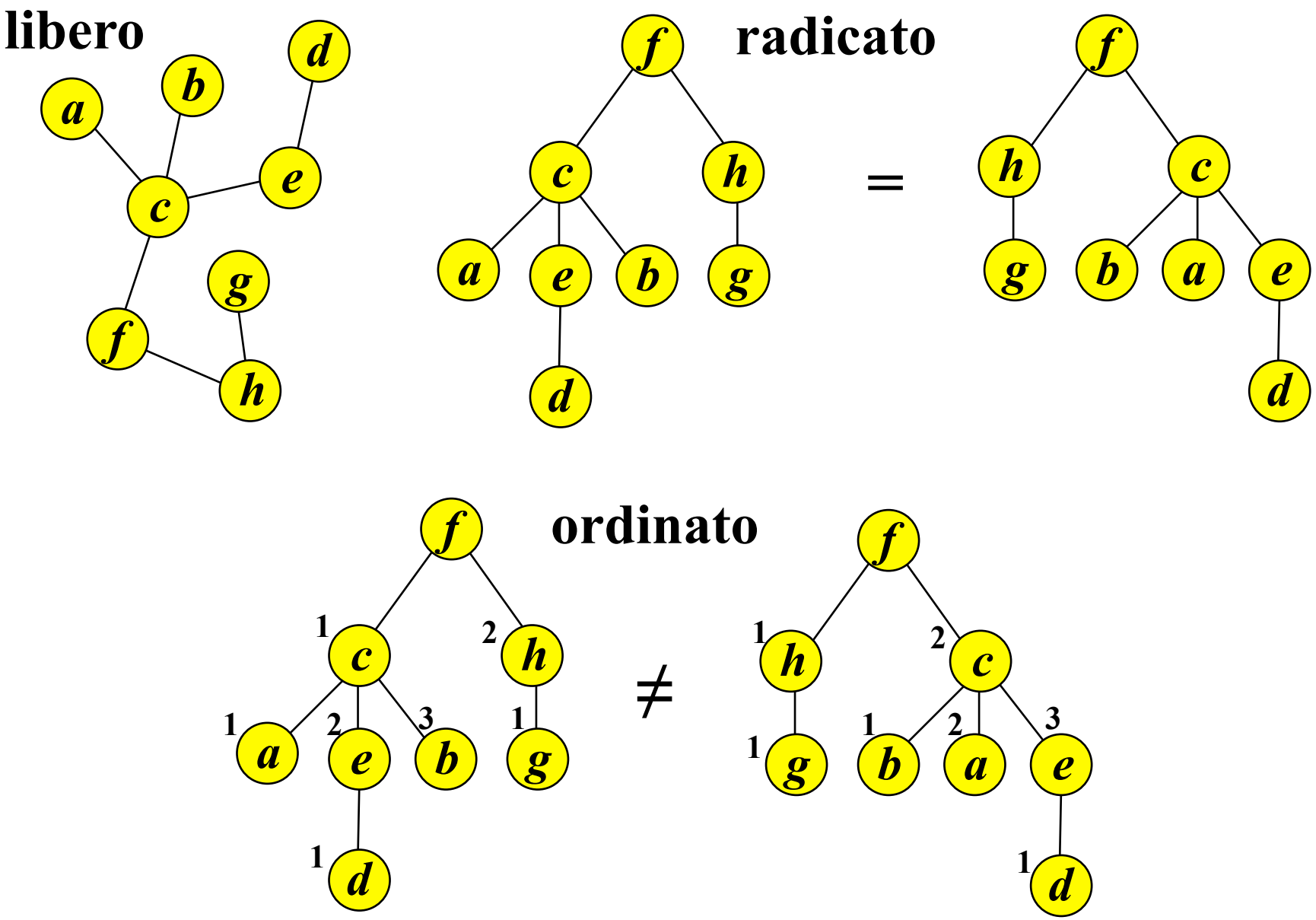
Alberi

Alberi liberi : grafi non orientati connessi e senza cicli.

Alberi radicati : alberi liberi in cui un vertice è stato scelto come radice.

Alberi ordinati : alberi radicati con un ordine tra i figli di un nodo.


libero
a
b
d
c
e
g
f
h
radicato
f
c
h
a
e
b
g
d
=
f
h
c
g
b
a
e
d
ordinato
f
1
c
2
h
1
a
2
e
3
b
1
g
1
d
≠
f
1
h
2
c
1
g
1
b
2
a
3
e
1
d

Alberi posizionali : alberi radicati in cui ad ogni figlio di un nodo è associata una posizione.

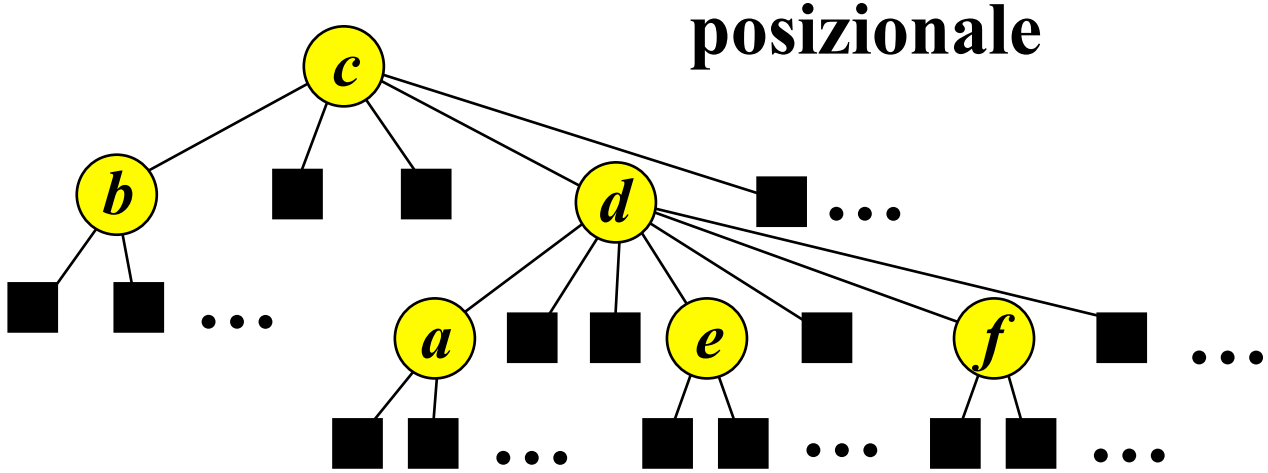
Le posizioni che non sono occupate da un nodo sono posizioni vuote (***nil***).

Alberi k-ari : alberi posizionali in cui ogni posizione maggiore di ***k*** è vuota.

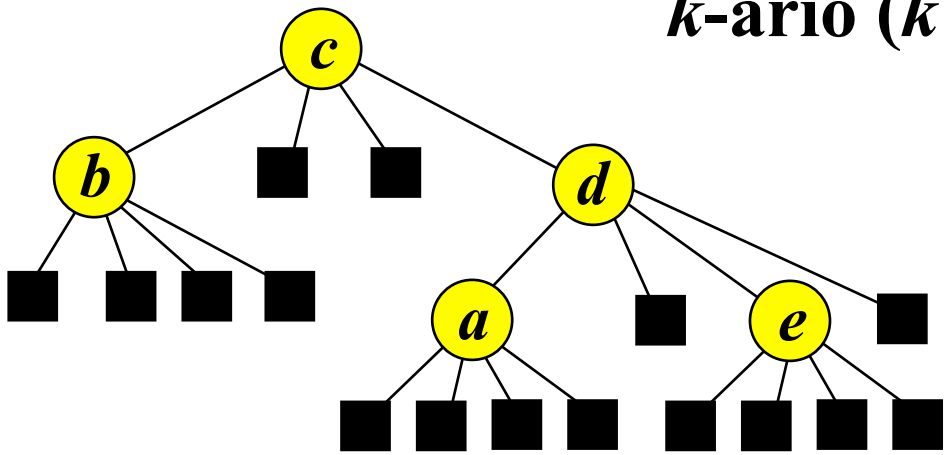
Alberi binari : alberi ***k***-ari con ***k* = 2.**

Il figlio in posizione **1** si dice ***figlio sinistro*** e quello in posizione **2** si dice ***figlio destro.***

posizionale



k-ario (*k* = 4)



Alberi binari

Il modo più conveniente per descrivere gli alberi binari è mediante la seguente.

Definizione ricorsiva di albero binario:

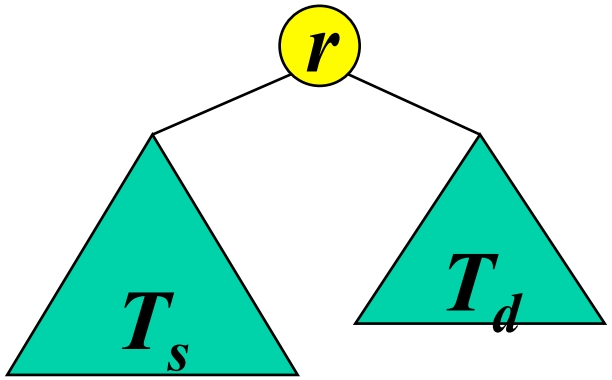
a) l'insieme vuoto $\varnothing$ è un albero binario;

b) se T_s e T_d sono alberi binari ed r è un *nodo* allora la terna ordinata (r, T_s, T_d) è un albero binario.

L'albero vuoto $\emptyset$ si rappresenta graficamente con quadratino nero

■

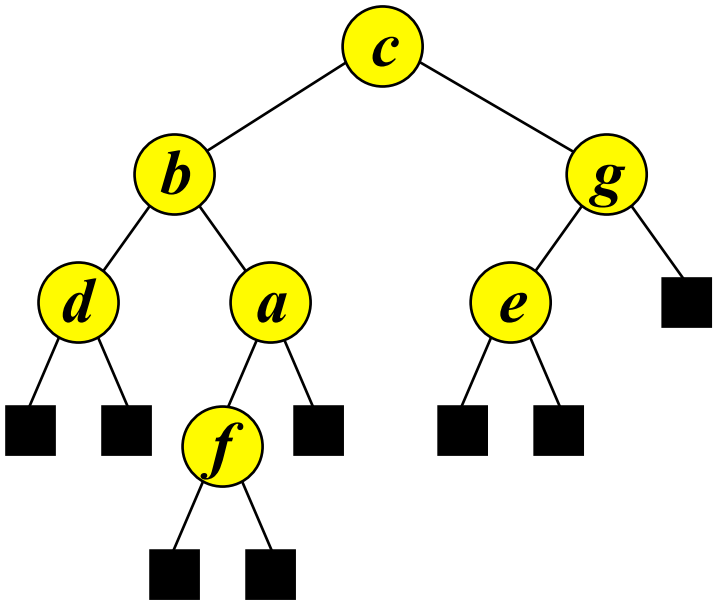
Per rappresentare l'albero $\boldsymbol{T = (r, T_s, T_d)}$ si disegna un nodo etichettato $\boldsymbol{r}$ e sotto di esso le due rappresentazioni dei sottoalberi $\boldsymbol{T_s}$ e $\boldsymbol{T_d}$, con $\boldsymbol{T_s}$ alla sinistra di $\boldsymbol{T_d}$



L'albero:

$$T = (c, (b, (d, \emptyset, \emptyset), (a, (f, \emptyset, \emptyset), \emptyset)), (g, (e, \emptyset, \emptyset), \emptyset))$$

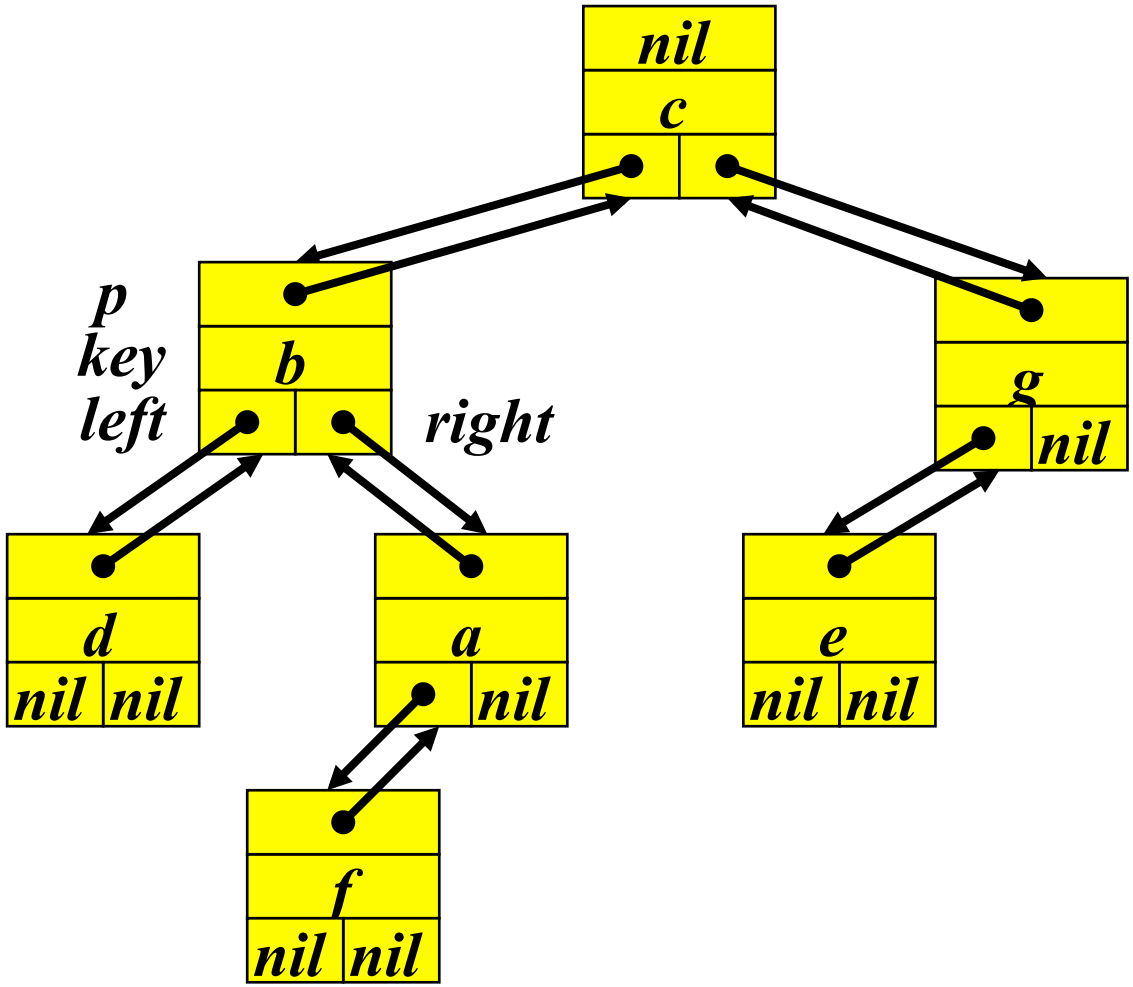
si rappresenta graficamente:



Nella memoria l'albero:

***T* = (*c*, (*b*, (*d*, Ø, Ø), (*a*, (*f*, Ø, Ø), Ø)), (*g*, (*e*, Ø, Ø), Ø))**

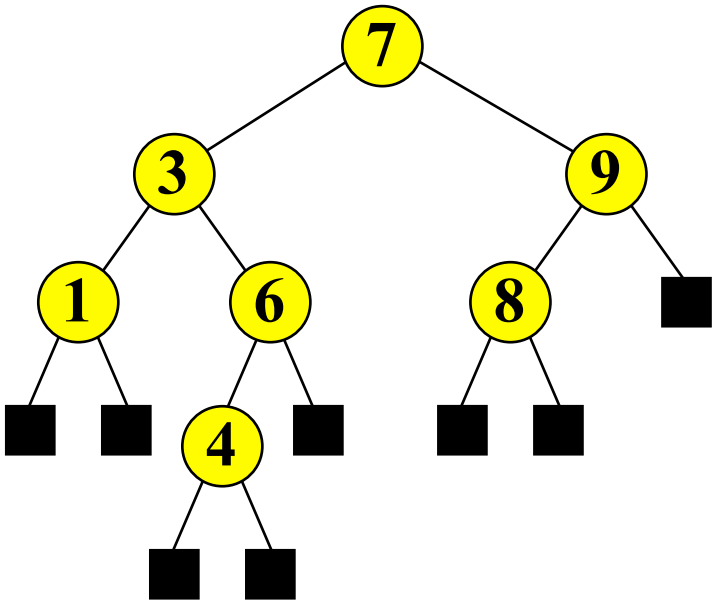
si rappresenta nel modo seguente:



Alberi binari di ricerca

Un albero binario di ricerca è un albero binario in cui la chiave di ogni nodo è maggiore o uguale delle chiavi dei nodi del sottoalbero sinistro e minore o uguale delle chiavi dei nodi del sottoalbero destro.

Ad esempio:



Operazioni sugli alberi binari di ricerca

Stampa della lista ordinata dei nodi:

```
Stampa(x)
  if x ≠ nil
      Stampa(x.left)
      print x
      Stampa(x.right)
```

Complessità:

$$T(0) = c$$

$$T(n) = T(k)+b+T(n-k-1)$$

Verifichiamo per sostituzione che

$$T(n) = (c + b)\, n + c$$

$$T(0) = c = (c + b)0 + c$$

$$T(n) = T(k) + b + T(n-k-1) =$$

$$= (c + b)k + c + b + (c + b)(n-k-1) + c$$

$$= (c + b)n + c$$

Ricerca di una chiave:

```
Search(x, k)
    if x == nil or k == x.key
        return x
    if k < x.key
        return Search(x.left, k)
    else
        return Search(x.right, k)
```

Complessità $O(h)$ dove h è l'altezza dell'albero.

Si può anche fare iterativa:

```
Search(x, k)
    while x ≠ nil and k ≠ x.key
        if k < x.key
                x = x.left
        else x = x.right
    return x
```

Complessità $O(h)$ dove h è l'altezza dell'albero.

Ricerca del minimo e del massimo:

```
Minimum(x)            // x ≠ nil
  while x.left ≠ nil
    x = x.left
  return x

Maximum(x)            // x ≠ nil
  while x.right ≠ nil
    x = x.right
  return x
```

Complessità $O(h)$ dove h è l'altezza dell'albero.

Ricerca di successivo e precedente

```
Successor(x)
    if x.right ≠ nil
        return Minimum(x.right)
    y = x.p
    while y ≠ nil and x == y.right
        x = y, y = y.p
    return y
```

Il precedente si ottiene cambiando ***right*** in ***left*** e ***Minimum*** in ***Maximum***.

Complessità $O(h)$ dove h è l'altezza dell'albero.

Inserzione di un nuovo elemento

```
Insert(T, z)                    // z.left = z.right = nil
  x = T.root, y = nil           // y padre di x
  while x ≠ nil                 // cerco dove mettere z
    y = x
    if z.key < y.key
        x = y.left
    else x = y.right
  z.p = y        // metto z al posto della foglia x
  if y == nil
     T.root = z
  elseif z.key < y.key
     y.left = z
  else y.right = z
```

Complessità ***O(h)*** dove ***h*** è l'altezza dell'albero.

Eliminazione di un elemento

Si riporta una versione semplificata, dove si spostano chiavi tra nodi diversi. Questo potrebbe rendere inconsistenti altri puntatori, a tali nodi.

A lezione, discussa una versione che non soffre di questo problema. Vedi Libro Paragrafo 12.3

Eliminazione di un elemento:

```
Delete(T, z)                                    // z ≠ nil
  if z.left == nil or z.right == nil   // tolgo z
    y = z          // che ha al più un solo figlio
  else // tolgo il successore di z che non ha
         // sottoalbero sinistro
    y = Successor(z), z.key = y.key
  // cerco l'eventuale unico figlio x di y
  if y.left == nil
    x = y.right
  else
    x = y.left
```

```
// metto x al posto di y
if x ≠ nil
    x.p = y.p
if y.p == nil
    T.root = x
elseif y == y.p.left
    y.p.left = x
else
    y.p.right = x
```

Complessità $O(h)$ dove h è l'altezza dell'albero.