# Alberi

E' una struttura gerarchica di ogni tipo

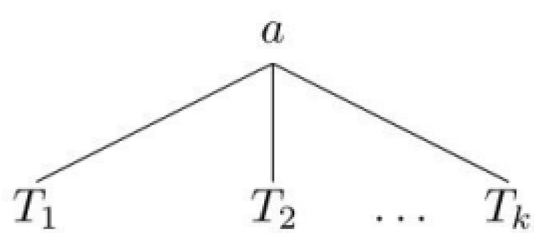
## Definizioni

Dato un insieme $A$ di etichette,l'insieme di alberi su $A$, denotato con $T(A)$, è definito induttivamente:

$$a \in A \land T_1 \in T(A) \land T_2 \in T(A) \land \cdots \land T_k \in T(A) \quad \text{con } k \geq 0$$

$$\Downarrow$$

$$\{a, T_1, T_2, ..., T_k\} \in T(A)$$

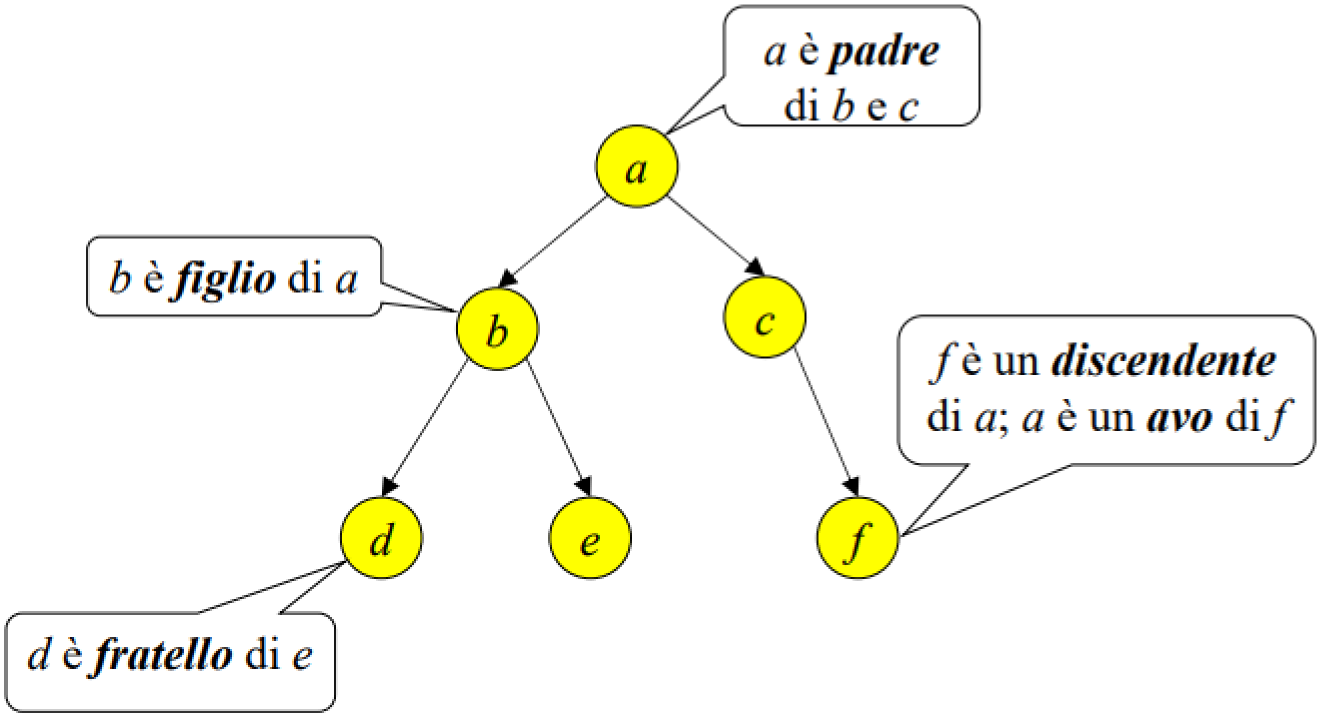



Un insieme di **alberi** viene detto **foresta**.
Inoltre un albero è un **grafo connesso aciclico**.

## Struttura albero

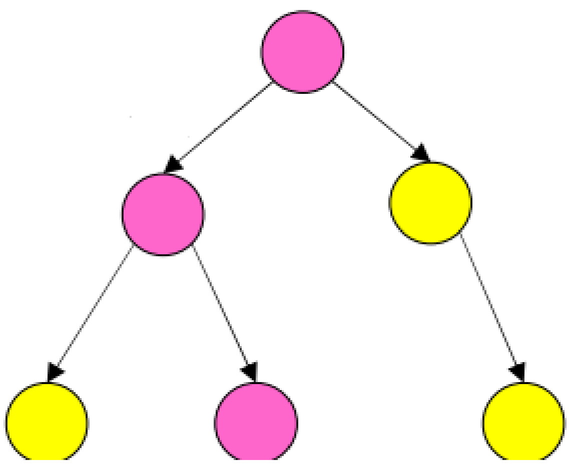
Ogni elemento di un albero viene detto **nodo**.
La **radice** è il nodo più in alto di tutti.
Una **foglia** è un qualsiasi nodo da cui non esce nessun **arco**.
Un nodo **non foglia** viene detto **interno**

## Parentele




## Cammino

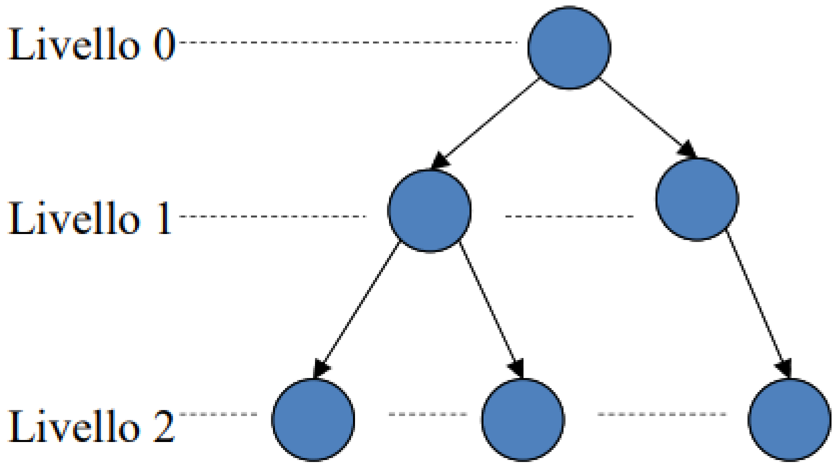
E' una sequenza di archi ciascuno incidente sul vertice di quello successivo.



Il cammino che parte dalla **radice** fino ad una **foglia** viene detto **ramo.**

## Livelli e grado

Il livello è l'insieme di vertici equidistanti dalla radice.




*Albero con 3 livelli. Ha un'altezza di 2*

Viene definita anche l'**altezza** come la massima distanza dalla radice di un livello non vuoto.

Per **grado** di un albero indichiamo il massimo numero di figli di un **nodo** qualunque.
L'albero **binario** ad esempio è un albero di grado 2.
Ogni nodo ha al massimo due figli.

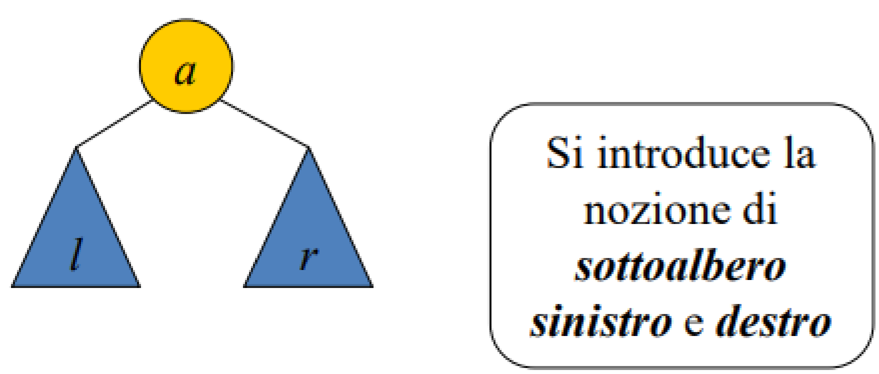
## Alberi binari posizionali

L'insieme degli alberi binari etichettati in $A$, $\mathrm{BT}(A)$, è definito induttivamente:

a) $\varnothing \in \mathrm{BT}(A)$ (albero vuoto)

b) $a \in A, l \in \mathrm{BT}(A), r \in \mathrm{BT}(A) \Rightarrow$

$$\{a, l, r\} \in \mathrm{BT}(A)$$




## Cardinalità alberi binari(e non)

Per **cardinalità** si intende il numero totale dei suoi nodi.

$\text{2-}\textsc{Tree-Card}(\text{2-Tree } T)$

**if** $T = nil$ **then**
    **return** $0$
**else**
    $l \leftarrow \text{2-}\textsc{Tree-Card}(T.left)$
    $r \leftarrow \text{2-}\textsc{Tree-Card}(T.right)$
    **return** $l + r + 1$
**end if**

Se l'albero non fosse binario, dovremmo ciclare tra tutti i nodi e calcolarci la cardinalità di ogni sotto-albero:

$k\text{-}\textsc{Tree-Card}(k\text{-Tree } T)$

**if** $T = nil$ **then**
    **return** $0$
**else**
    $card \leftarrow 1$
    $C \leftarrow T.child$
    **while** $C \neq nil$ **do**
        $card \leftarrow card + k\text{-}\textsc{Tree-Card}(C)$
        $C \leftarrow C.sibling$
    **end while**
    **return** $card$
**end if**

*Calcolo della cardinalità di un albero k-ario*

## Altezza alberi binari(e non)

**Alberi binari:**

```
2-TREE-HIGHT(2-Tree T)
if T.left = nil and T.right = nil then
    return 0        ▷ T ha un solo nodo
else
    hl, hr ← 0
    if T.left ≠ nil then
        hl ← 2-TREE-HIGHT(T.left)
    end if
    if T.right ≠ nil then
        hr ← 2-TREE-HIGHT(T.right)
    end if
    return 1 + max{hl, hr}
end if
```

**Alberi k-ari:**

```
kTREE-HIGHT(T)
if T.child = nil then
    return 0        ▷ T ha un solo nodo
else
    h ← 0
    C ← T.child
    while C ≠ nil do
        h ← max{h, kTREE-HIGHT(C)}
        C ← C.sibling
    end while
    return h + 1
end if
```

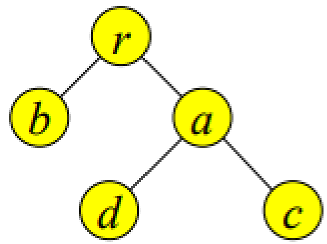
*Il meccanismo è analogo al calcolo della cardinalità per alberi k-ari*

## Visite

La visita(completa) di un albero consiste in un'ispezione dei nodi dell'albero in cui ciascun nodo viene *visitato* esattamente una volta.

**Visita in profondità(DFS):** Avviene lungo i **rami** dalla radice alle foglie
**Visita in ampiezza(BFS):** Avviene lungo i **livelli**, da quello della radice in poi

$T$




DFS con preordine destro di T: r, a, c, d, b

DFS con preordine sinistro di T: r, b, a, d, c

BFS con livelli da sinistra a destra di T: r, b, a, d, c

BFS con livelli da destra a sinistra di T: r, a, b, c, d

*Esempio di varie visite su un albero T*

### DFS (Depth-First-Search)

**Ricorsivo:**

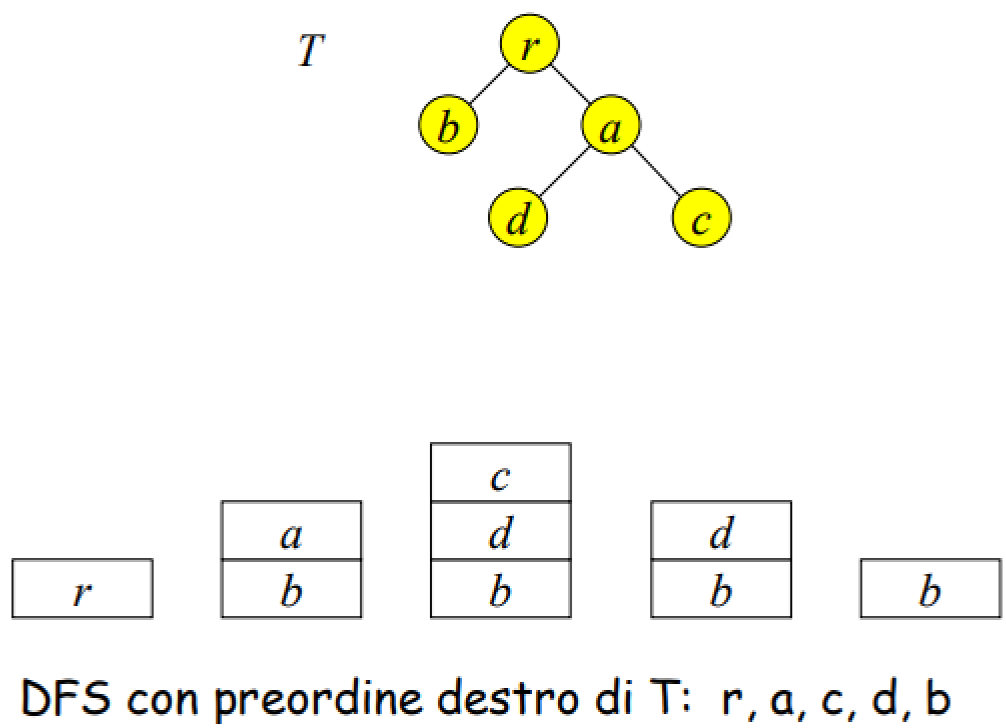
$\textsc{Tree-DFS}(k\text{-Tree } T)$
visita $T.key$
$C \leftarrow T.child$
**while** $C \neq nil$ **do**
    $\textsc{Tree-DFS}(C)$
    $C \leftarrow S.sibling$
**end while**

*Visita il nodo, e poi cicla tra tutti i sibling del figlio*

**Con ausilio di pila:**

```
TREE-DFS-STACK(k-Tree T)
S ← pila vuota
Push(S, T)
while S ≠ la pila vuota do
    T' ← Pop(S)
    visita T'.key
    for all C figlio di T' do
        Push(S, C)
    end for
end while
```

*Il meccanismo è uguale a quello ricorsivo, tranne per il fatto che in questo caso l'algoritmo si appoggia ad una pila. Viene fatto il push dei sibiling del nodo. Una volta che ho finito di esaminare i figli del nodo di partenza, faccio il pop di quest'ultimo.*

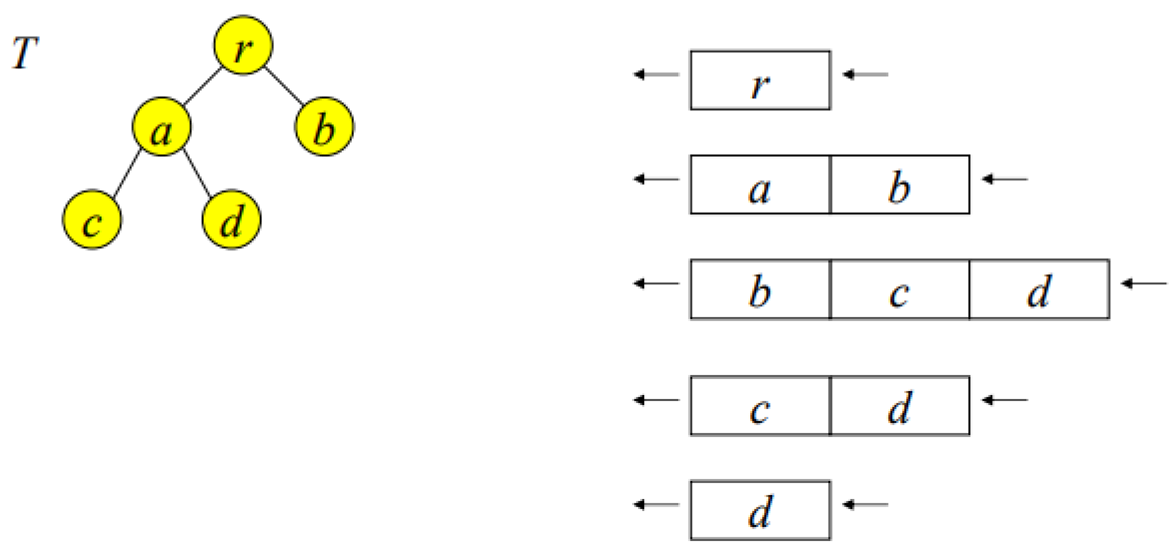



*Esempio di DFS con ausilio di pila*

### BFS (Breatdh-First-Search)

```
TREE-BFS(k-Tree T)
Q ← coda vuota
Enqueue(Q, T)
while Q ≠ la coda vuota do
    T' ← Dequeue(Q)
    visita T'.key
    for all C figlio di T' do
        Enqueue(Q, C)
    end for
end while
```

*Metto in coda tutti i figlio del nodo, li visito nell'ordine in cui sono stati incodati e li tolgono dalla coda. Itero sto procedimento per tutti i figli dei figli etc..*




BFS con livelli da sinistra a destra di T: r, a, b, c, d

*Esempio di BFS con coda*

## Complessità delle visite

Sia la dimensione $n$ la cardinalità dell'albero.
Possiamo limitarci a contare quante operazione Push/Enqueue e Pop/Dequeue vengono eseguite.
Dal momento che ogni nodo dell'albero viene inserito ed estratto esattamente una volta dalla coda/stack, possiamo concludere che questi algoritmi hanno costo $O(2N) = O(N)$