Alberi

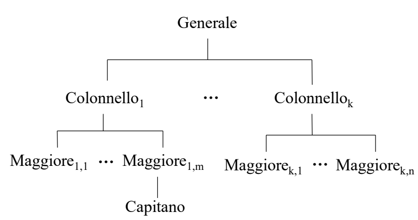
Algoritmi e strutture dati

Ugo de'Liguoro, Andras Horvath

1

Cosa sono gli alberi?

Strutture gerarchiche di ogni tipo:



2

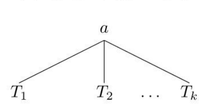
Definizione

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

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

$$\Downarrow$$

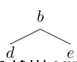
$$\{a, T_1, T_2, ..., T_k\} \in T(A)$$



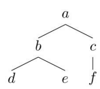
3

Utilizzo della definizione

- sia $A = \{a, b, c, d, e, f\}$
- allora $\{a\}, \{b\}, \{c\}, \{d\}, \{e\}, \{f\}$ sono alberi che fanno parte di $T(A)$ (contengono un nodo solo)
- allora $\{b, \{d\}, \{e\}\}$ e $\{c, \{f\}\}$ sono alberi che fanno parte di $T(A)$



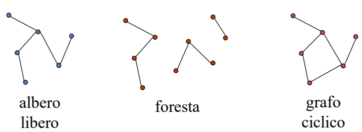
- allora $\{a, \{b, \{d\}, \{e\}\}, \{c, \{f\}\}\}$ è un albero che fanno parte di $T(A)$



4

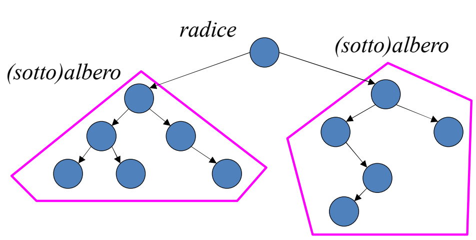
Alberi come grafi

- un ***albero*** è un grafo connesso aciclico
- un insieme di alberi è una ***foresta***



5

Alberi come grafi



6

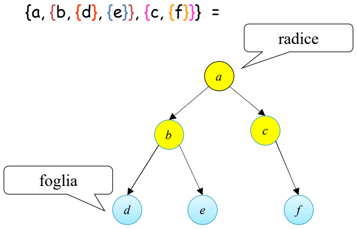
Alberi radicati

- la ***radice*** è un nodo privilegiato di un albero
- una ***foglia*** è un nodo da cui non esce alcun arco
- un nodo che non sia una foglia si dice ***interno***

7

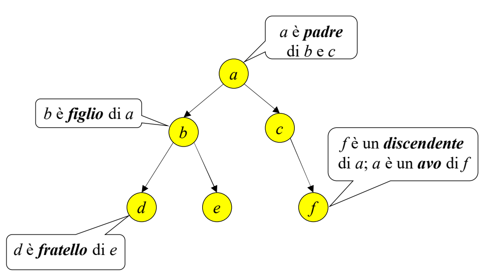
Alberi come grafi

{a, {b, {d}, {e}}, {c, {f}}} =



8

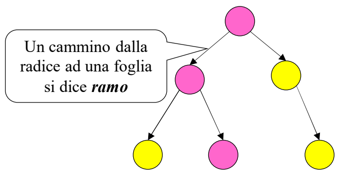
Parentele



9

Cammini

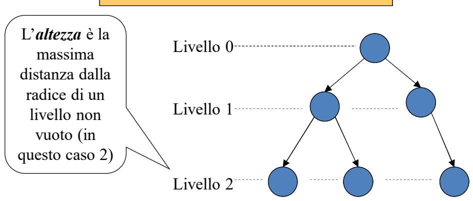
Cammino: sequenza di archi ciascuno incidente sul vertice di quello successivo



10

Livelli

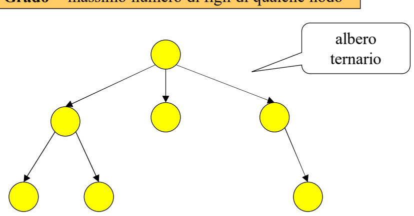
Livello: insieme di vertici equidistanti dalla radice



11

Alberi di grado *k* (*k*-ari)

Grado = massimo numero di figli di qualche nodo

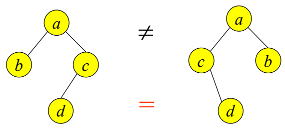


12

Alberi ordinati

Un albero è ***ordinato*** quando lo sono (linearmente) i suoi livelli.

Come alberi non ordinati sono =, cioè uguali.



Come alberi ordinati sono ≠, cioè diversi.

In ogni caso i sottoalberi che hanno radice in *c* sono uguali (si tratta di alberi non posizionali, vedi lucido successivo).

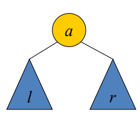
13

Alberi binari posizionali

L'insieme degli alberi binari etichettati in A, BT(A), è definito induttivamente:

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

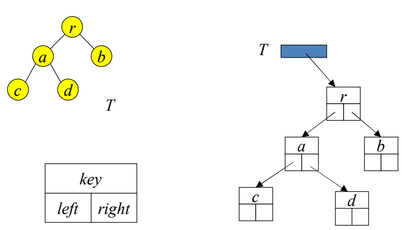
b) $a \in A,\ l \in \mathrm{BT}(A),\ r \in \mathrm{BT}(A) \Rightarrow \{a, l, r\} \in \mathrm{BT}(A)$



Si introduce la nozione di ***sottoalbero sinistro*** e ***destro***

14

Alberi binari realizzati con puntatori



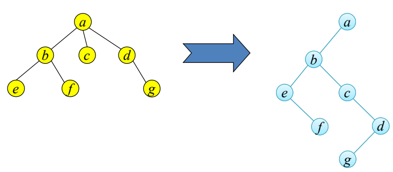
15

Alberi *k*-ari realizzati con puntatori

- per rappresentare un alberi *k*-ario in ogni nodo:
  - etichetta (key)
  - *k* puntatori
- bisogna sapere *k* a priori e i nil possono occupare tanta memoria
- come alternativa in ogni nodo si può avere:
  - etichetta (key)
  - una lista di puntatori

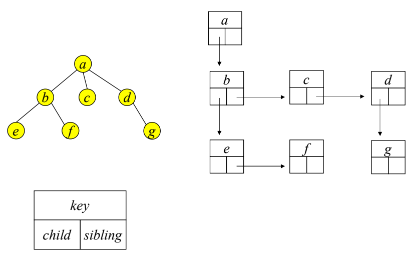
16

Codifica binaria di alberi *k*-ari



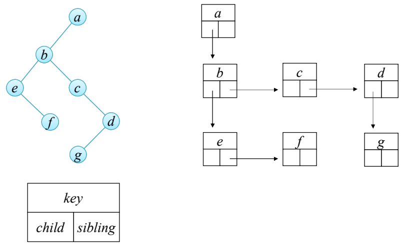
17

Codifica binaria di alberi *k*-ari



18

Codifica binaria di alberi *k*-ari



19

Cardinalità, alberi binari

La **cardinalità** di un albero è il numero dei suoi nodi.

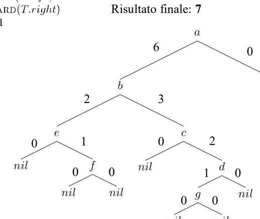
```
2-TREE-CARD(2-Tree T)
if T = nil then
    return 0
else
    l ← 2-TREE-CARD(T.left)
    r ← 2-TREE-CARD(T.right)
    return l + r + 1
end if
```

20

Cardinalità, alberi binari

```
2-TREE-CARD(2-Tree T)
if T = nil then
    return 0
else
    l ← 2-TREE-CARD(T.left)
    r ← 2-TREE-CARD(T.right)
    return l + r + 1
end if
```

Risultato finale: **7**



21

Cardinalità, alberi *k*-ari

La **cardinalità** di un albero è il numero dei suoi nodi.

```
k-Tree-Card(k-Tree T)
if T = nil then
    return 0
else
    card ← 1
    C ← T.child
    while C ≠ nil do
        card ← card + k-Tree-Card(C)
        C ← C.sibling
    end while
    return card
end if
```

Visto che usiamo la rappresentazione binaria funzionerebbe anche l'algoritmo precedente sostituendo *left* con *child* e *right* con *sibling*.

22

Altezza, alberi binari

```
2-Tree-Hight(2-Tree T)        ▷ pre: T non è vuoto
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
```

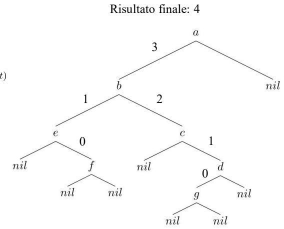
L'altezza è il massimo dei livelli, ossia il massimo delle lunghezze dei rami

23

Altezza, alberi binari


```
2-Tree-Hight(2-Tree T)        ▷ pre: T non è vuoto
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
```


Risultato finale: 4



24

Altezza, alberi *k*-ari

kTREE-HIGHT(T) ▷ pre: T non è vuoto
if $T.child = nil$ **then**
    return 0 ▷ T ha un solo nodo
else
    $h \leftarrow 0$
    $C \leftarrow T.child$
    while $C \neq nil$ **do**
        $h \leftarrow \max\{h, k\text{TREE-HIGHT}(C)\}$
        $C \leftarrow C.sibling$
    end while
    return $h + 1$
end if

L'altezza è il massimo dei livelli, ossia il massimo delle lunghezze dei rami

25

Visite

Qual è la complessità di questi algoritmi?

Per rispondere osserviamo che hanno tutti la struttura di una visita!

26

Visite

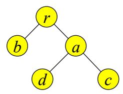
La ***visita*** (completa) di un albero consiste in un'ispezione dei nodi dell'albero in cui ciascun nodo sia "visitato" (ispezionato) esattamente una volta.

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

27

Varie visite DFS e BFS

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

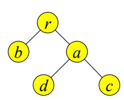
BFS con livelli da destra a sinistra di T: r, a, b, c, d

28

DFS (ricorsiva)

```
TREE-DFS(k-Tree T)
visita T.key
C ← T.child
while C ≠ nil do
    TREE-DFS(C)
    C ← S.sibling
end while
```

T



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

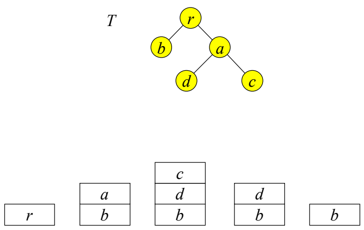
29

DFS con l'ausilio di una pila

```
TREE-DFS-STACK(k-Tree T)          ▷ pre: T non è vuoto
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
```

30

DFS con l’ausilio di una pila



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

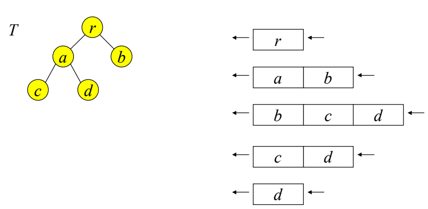
31

Visita in ampiezza (BFS)

```
TREE-BFS(k-Tree T)        ▷ pre: T non è vuoto
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
```

32

BFS con l’ausilio di una coda



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

33

DFS versus BFS

```
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
```

```
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
```

34

Complessità delle visite

- la dimensione n di un albero è la sua cardinalità
- per limitare il tempo possiamo contare quante operazioni Push/Pop ovvero Enqueue/Dequeue avvengono in una DFS o BFS
- ogni nodo dell'albero viene inserito ed estratto esattamente una volta
- dunque DFS e BFS hanno costo $O(2n) = O(n)$

35