

# Algoritmi e Strutture Dati

## Capitolo 3

### Strutture dati elementari

# Gestione di collezioni di oggetti

## Tipo di dato:

- Specifica una collezione di oggetti e delle operazioni di interesse su tale collezione (es. inserisci, cancella, cerca)

## Struttura dati:

- Organizzazione dei dati che permette di memorizzare la collezione e supportare le operazioni di un tipo di dato usando meno risorse di calcolo possibile

# Il tipo di dato Dizionario

**tipo** Dizionario:

**dati:**

un insieme  $S$  di coppie ( $elem$ ,  $chiave$ ).

**operazioni:**

$insert(elem\ e, chiave\ k)$

aggiunge a  $S$  una nuova coppia  $(e, k)$ .

$delete(chiave\ k)$

cancella da  $S$  la coppia con chiave  $k$ .

$search(chiave\ k) \rightarrow elem$

se la chiave  $k$  è presente in  $S$  restituisce l'elemento  $e$  ad essa associato, e null altrimenti.

# Il tipo di dato Pila

**tipo** Pila:

**dati:**

una sequenza  $S$  di  $n$  elementi.

**operazioni:**

$\text{isEmpty}() \rightarrow \text{result}$

restituisce `true` se  $S$  è vuota, e `false` altrimenti.

$\text{push}(\text{elem } e)$

aggiunge  $e$  come ultimo elemento di  $S$ .

$\text{pop}() \rightarrow \text{elem}$

toglie da  $S$  l'ultimo elemento e lo restituisce.

$\text{top}() \rightarrow \text{elem}$

restituisce l'ultimo elemento di  $S$  (senza toglierlo da  $S$ ).

# Il tipo di dato Coda

**tipo** Coda:

**dati:**

una sequenza  $S$  di  $n$  elementi.

**operazioni:**

`isEmpty()`  $\rightarrow$  *result*

restituisce `true` se  $S$  è vuota, e `false` altrimenti.

`enqueue(elem e)`

aggiunge  $e$  come ultimo elemento di  $S$ .

`dequeue()`  $\rightarrow$  *elem*

toglie da  $S$  il primo elemento e lo restituisce.

`first()`  $\rightarrow$  *elem*

restituisce il primo elemento di  $S$  (senza toglierlo da  $S$ ).

# Tecniche di rappresentazione dei dati

## Rappresentazioni indicizzate:

- I dati sono contenuti (principalmente) in array

## Rappresentazioni collegate:

- I dati sono contenuti in record collegati fra loro mediante puntatori

## Rappresentazioni indicizzate:

- **Array:** collezione di celle numerate che contengono elementi di un tipo prestabilito

**Proprietà (forte):** gli indici delle celle di un array sono numeri consecutivi

**Proprietà (debole):** non è possibile aggiungere nuove celle ad un array

## Rappresentazioni collegate:

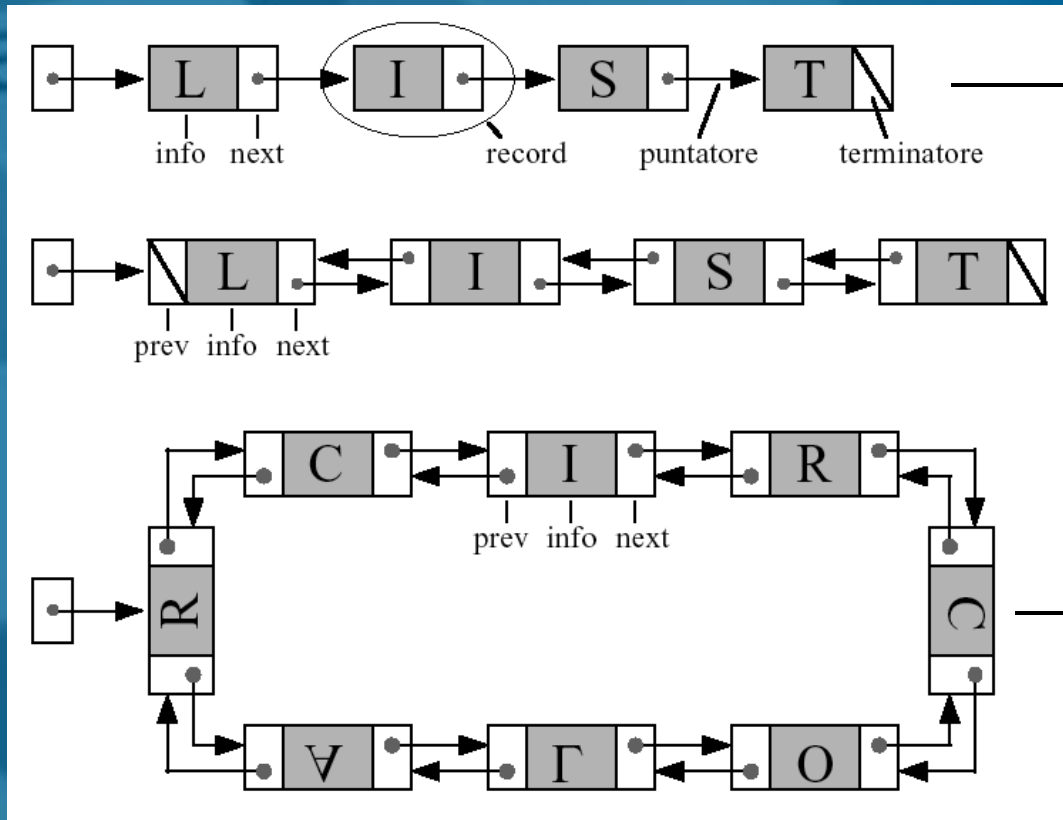
- i costituenti di base sono i *record*
- i record sono numerati tipicamente con il loro *indirizzo di memoria*
- record creati e distrutti individualmente e dinamicamente
- il collegamento tra un record A e un record B è realizzato tramite un *puntatore*

**Proprietà (forte):** è possibile aggiungere o togliere record a una struttura collegata

**Proprietà (debole):** gli indirizzi dei record di una struttura collegata non sono necessariamente consecutivi



# Esempi di strutture collegate



Lista semplice

Lista doppiamente collegata

Lista circolare doppiamente collegata

# Pro e contro

## Rappresentazioni indicizzate:

- **Pro:** accesso diretto ai dati mediante indici
- **Contro:** dimensione fissa (riallocazione array richiede tempo lineare)

## Rappresentazioni collegate:

- **Pro:** dimensione variabile (aggiunta e rimozione record in tempo costante)
- **Contro:** accesso sequenziale ai dati

# realizzazione di un dizionario

Metodo più semplice: **array non ordinato** (sovradimensionato)

**Insert** → costa  $O(1)$  – inserisco dopo ultimo elemento

**Search** → costa  $O(n)$  – devo scorrere l'array

**Delete** → costa  $O(n)$  – delete = search + cancellazione

## Array ordinato:

**Search** →  $O(\log(n))$  – ricerca binaria

**Insert** →  $O(n)$

Ho bisogno di:

$O(\log(n))$  confronti → per trovare la giusta posizione in cui inserire l'elemento

$O(n)$  trasferimenti → per mantenere l'array ordinato

(Ricorda che  $O(n) + O(\log(n)) = O(n)$ )

**Delete** →  $O(n)$  (come per **Insert**)

## ...e con le liste?

### Lista non Ordinata

Search –  $O(n)$

Insert –  $O(1)$

Delete –  $O(n)$

### Lista Ordinata

Search –  $O(n)$  non posso usare la ricerca binaria

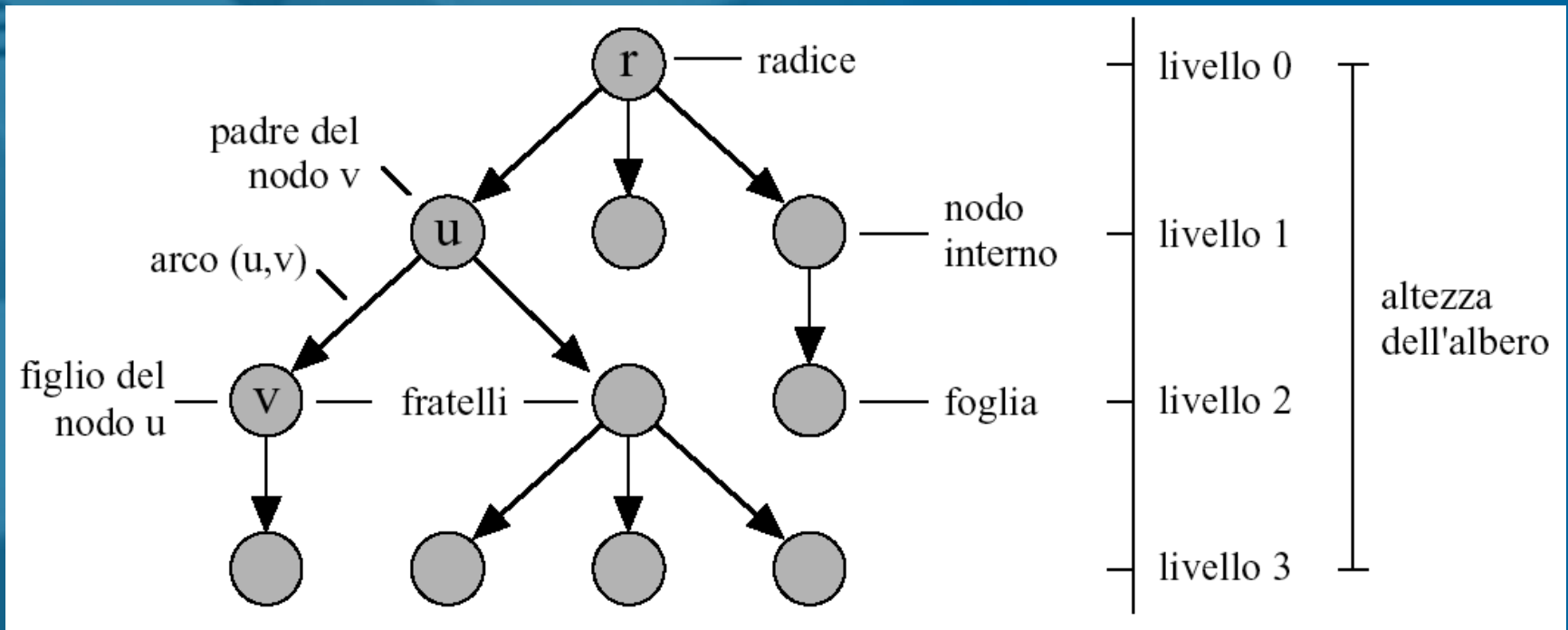
Insert –  $O(n)$  devo mantenere ordinata la lista

Delete –  $O(n)$

# Esercizi

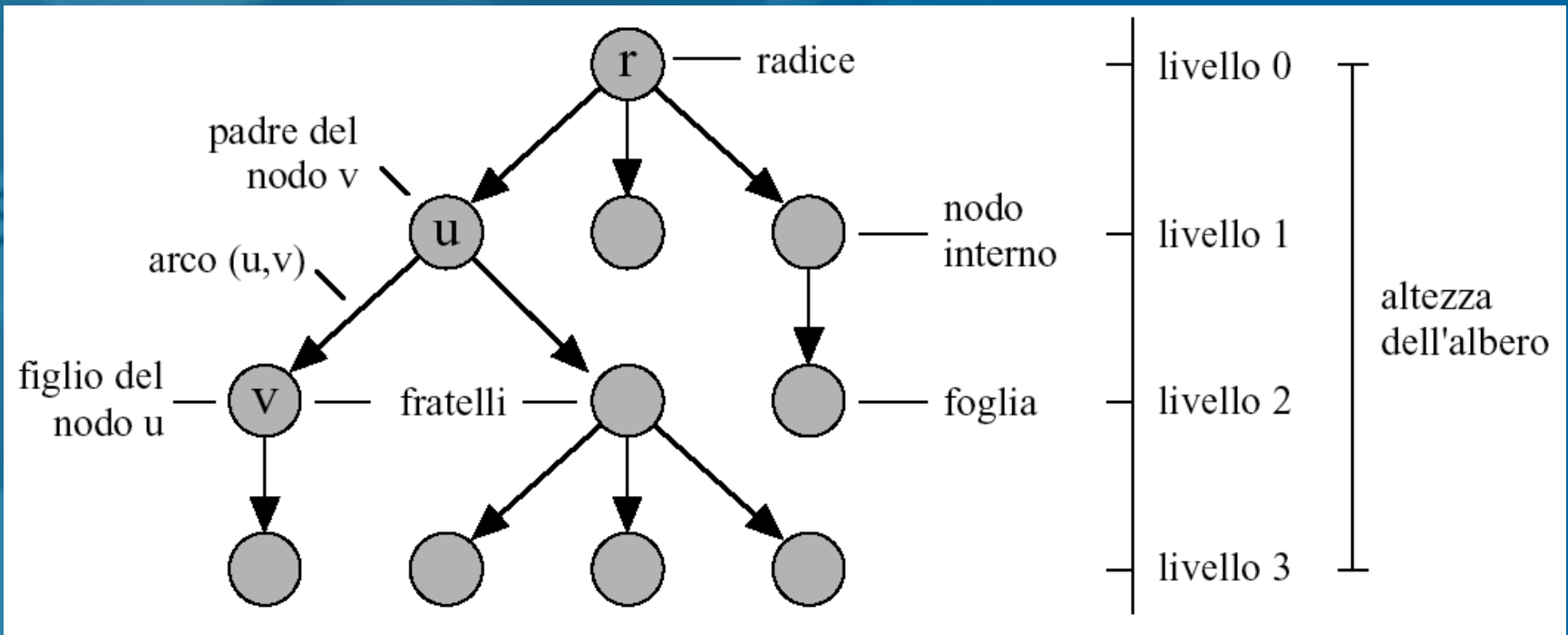
1. Progettare una struttura dati **indicizzata** che implementi il tipo di dato *Pila* e il tipo di dato *Coda*. Le operazioni devono avere complessità temporale costante.
2. Progettare una struttura dati **collegata** che implementi il tipo di dato *Pila* e il tipo di dato *Coda*. Le operazioni devono avere complessità temporale costante.

## Organizzazione gerarchica dei dati



Dati contenuti nei **nodi**, relazioni gerarchiche definite dagli **archi** che li collegano

# Alberi: altre definizioni



**grado di un nodo**: numero dei suoi figli

**albero d-ario, albero d-ario completo**

u **antenato** di v se u è raggiungibile da v risalendo di padre in padre

v **discendente** di u se u è un antenato di v

# Rappresentazioni indicizzate di alberi

**Idea:** ogni cella dell'array contiene

- le informazioni di un nodo
- eventualmente altri indici per raggiungere altri nodi

## Vettore dei padri

Per un albero con  $n$  nodi uso un array  $P$  di dimensione (almeno)  $n$

Una generica cella  $i$  contiene una coppia (**info**,**parent**), dove:

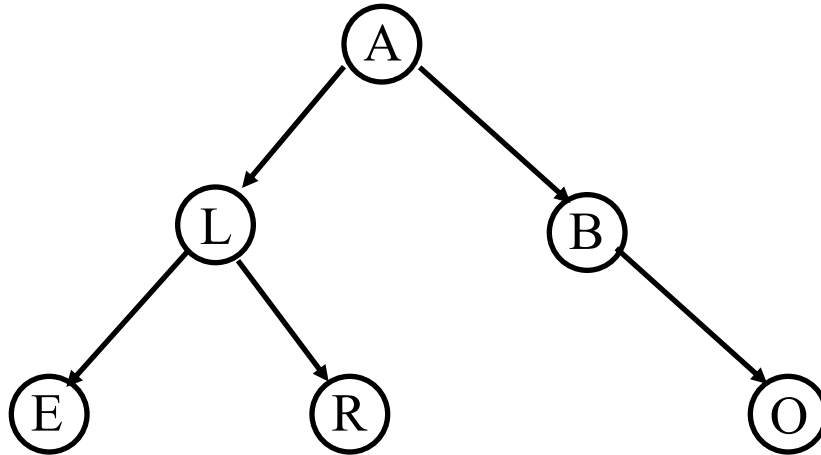
**info:** contenuto informativo del nodo  $i$

**parent:** indice (nell'array) del nodo padre di  $i$

**Vettore posizionale** (per alberi  $d$ -ari (quasi) completi)



## Vettore dei padri: un esempio



**P**

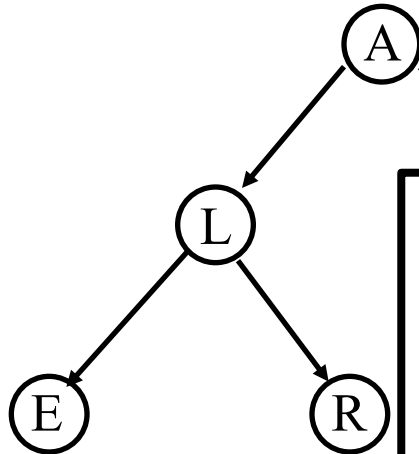
(L,3)	(B,3)	(A,null)	(O,2)	(E,1)	(R,1)
1	2	3	4	5	6

$(P[i].info, P[i].parent)$

$P[i].info$ : contenuto informativo nodo

$P[i].parent$ : indice del nodo padre

## Vettore dei padri: un esempio



### Osservazioni:

-# arbitrario di figli

-tempo per individuare il padre di un nodo:  $O(1)$

-tempo per individuare uno o più figli di un nodo:  $O(n)$

**P**

(L,3)	(B,3)	(A,null)	(O,2)	(E,1)	(R,1)
1	2	3	4	5	6

$(P[i].info, P[i].parent)$

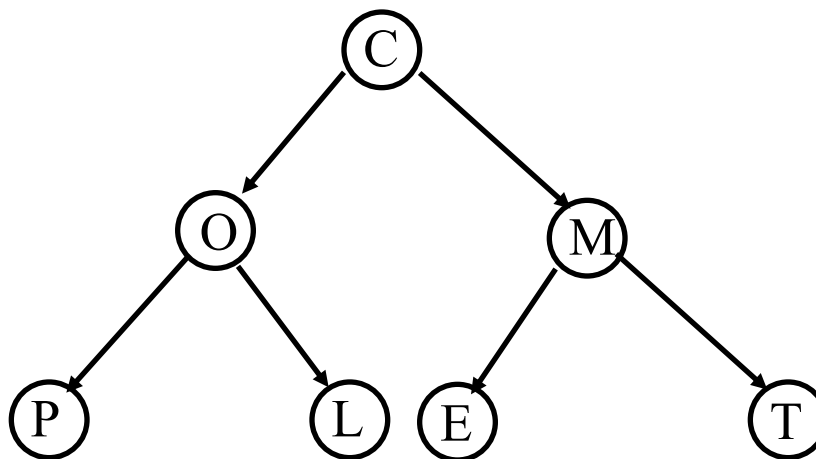
$P[i].info$ : contenuto informativo nodo

$P[i].parent$ : indice del nodo padre

## Vettore posizionale (per alberi $d$ -ari completi)

- nodi arrangiati nell'array “per livelli”
- $j$ -esimo figlio ( $j \in \{1, \dots, d\}$ ) di  $i$  è in posizione  $d(i-1)+j+1$
- il padre di  $i$  è in posizione  $\lfloor (i-2)/d \rfloor + 1$

$d = 2$

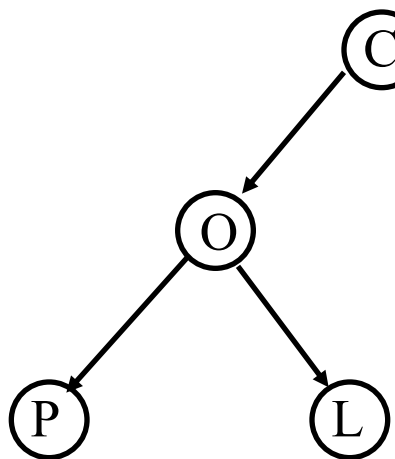


A

C	O	M	P	L	E	T
1	2	3	4	5	6	7

## Vettore posizionale (per alberi $d$ -ari completi)

- nodi arrangiati nell'array “per livelli”
- $j$ -esimo figlio ( $j \in \{1, \dots, d\}$ ) di  $i$  è in posizione  $d(i-1)+j+1$
- il padre di  $i$  è in posizione  $\lfloor (i-2)/d \rfloor + 1$



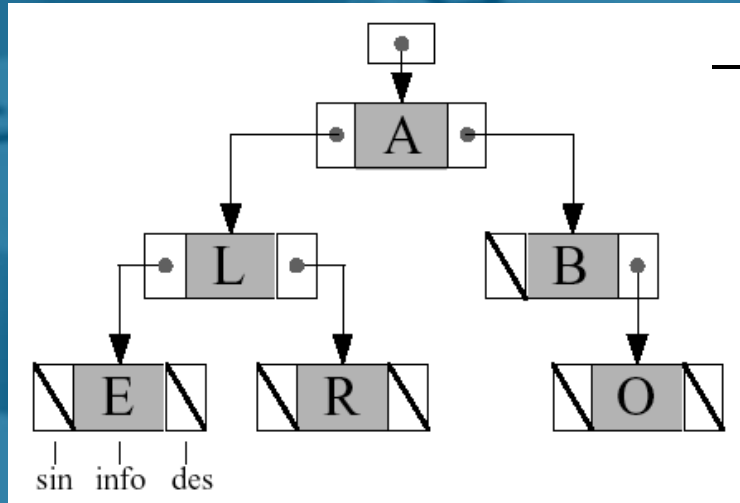
### Osservazioni:

- # di figli esattamente  $d$
- solo per alberi completi o quasi completi
- tempo per individuare il padre di un nodo:  $O(1)$
- tempo per individuare uno specifico figlio di un nodo:  $O(1)$

A

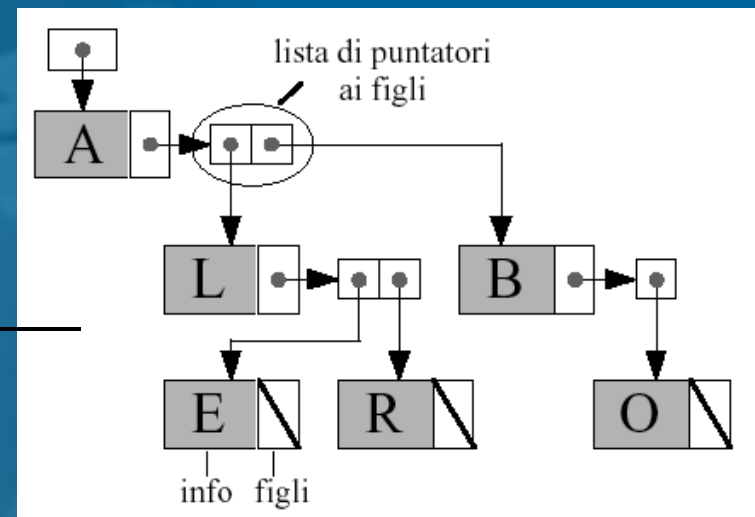
C	O	M	P	L	E	T
1	2	3	4	5	6	7

# Rappresentazioni collegate di alberi

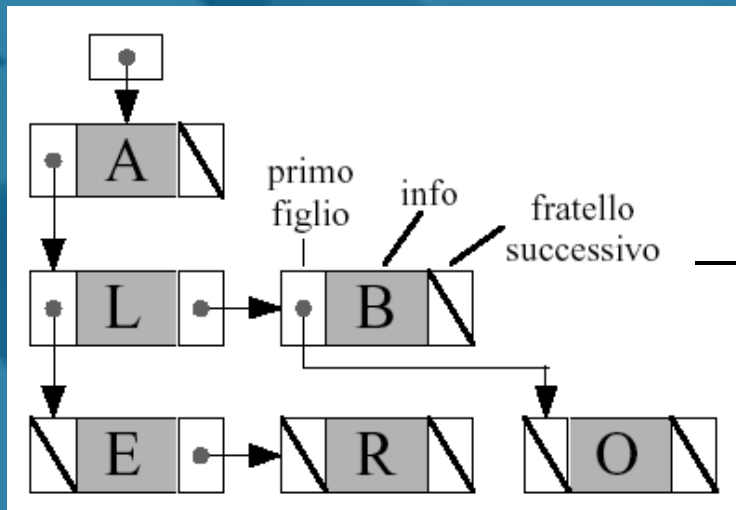


Rappresentazione con puntatori ai figli (nodi con numero limitato di figli)

Rappresentazione con liste di puntatori ai figli (nodi con numero arbitrario di figli)



# Rappresentazioni collegate di alberi



Rappresentazione di tipo primo figlio-fratello successivo (nodi con numero arbitrario di figli)

Tutte le rappresentazioni viste possono essere arricchite per avere in ogni nodo anche un puntatore al padre

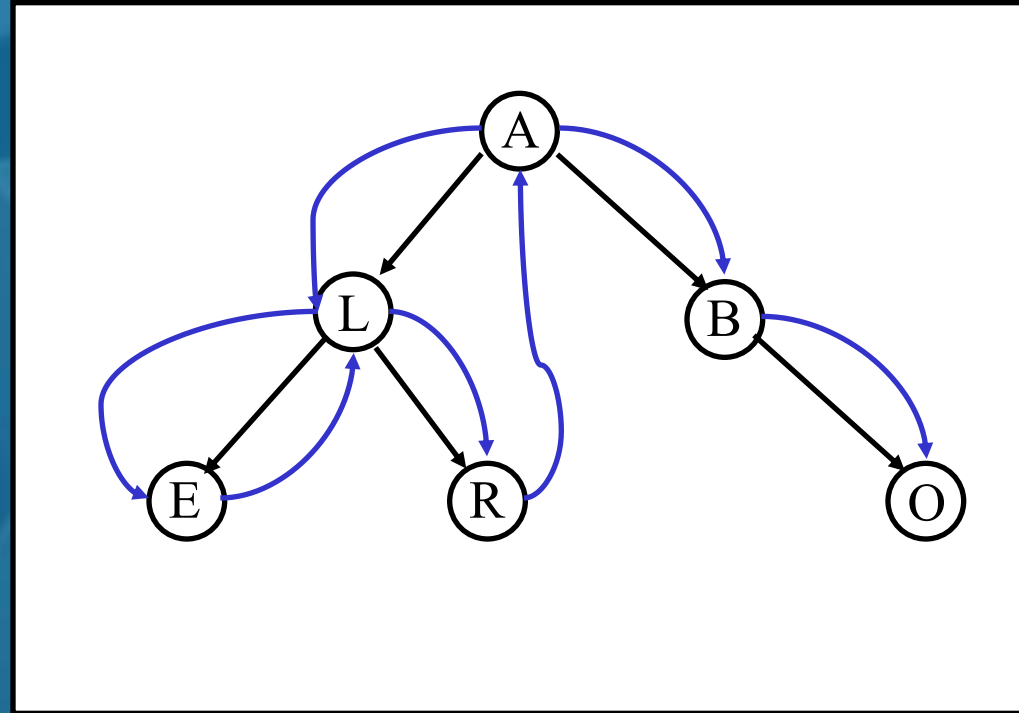
# Visite di alberi

Algoritmi che consentono l'**accesso sistematico ai nodi e agli archi** di un albero

Gli algoritmi di visita si distinguono in base al particolare ordine di accesso ai nodi

# Algoritmo di visita in profondità

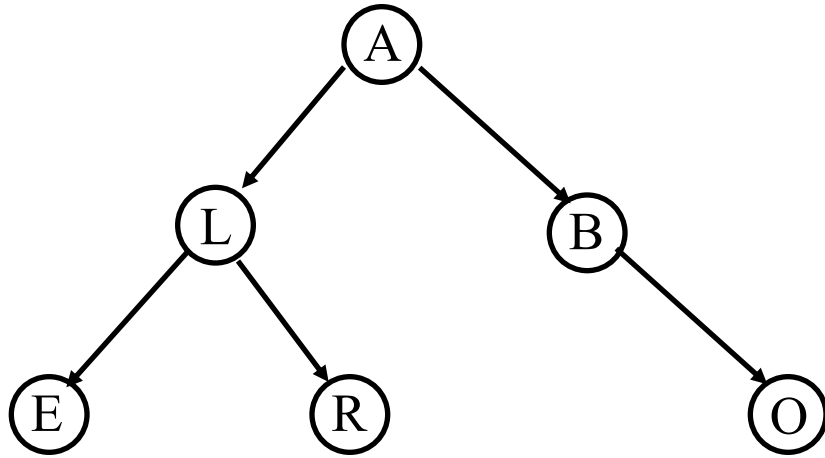
L'algoritmo di visita in profondità (DFS) parte da r e procede visitando nodi di figlio in figlio fino a raggiungere una foglia. Retrocede poi al primo antenato che ha ancora figli non visitati (se esiste) e ripete il procedimento a partire da uno di quei figli.





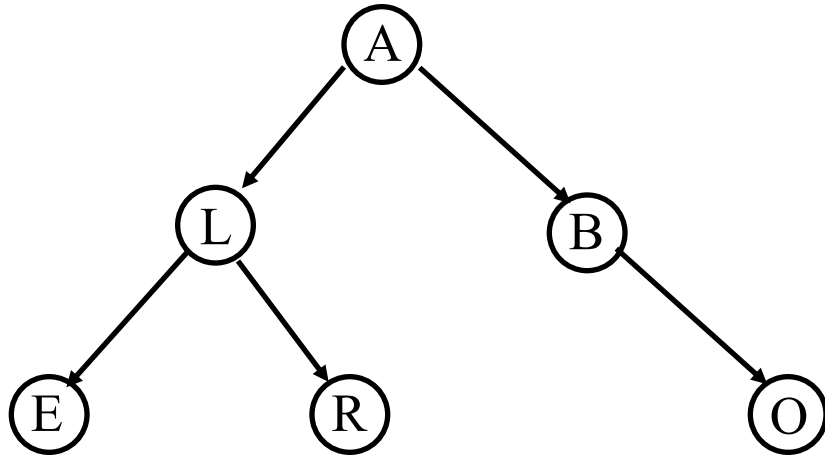
# Algoritmo di visita in profondità (per alberi binari)

```
algoritmo visitaDFS(nodo  $r$ )  
  Pila  $S$   
   $S.push(r)$   
  while (not  $S.isEmpty()$ ) do  
     $u \leftarrow S.pop()$   
    if ( $u \neq null$ ) then  
      visita il nodo  $u$   
       $S.push(\text{figlio destro di } u)$   
       $S.push(\text{figlio sinistro di } u)$ 
```



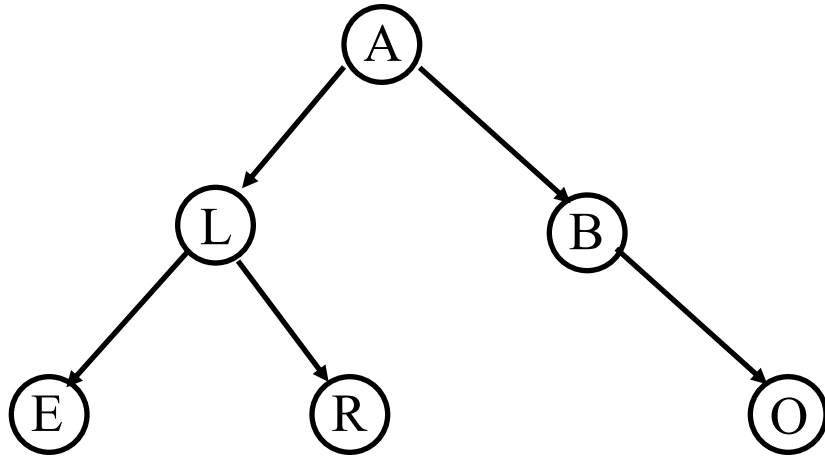
Ordine di visita:

A



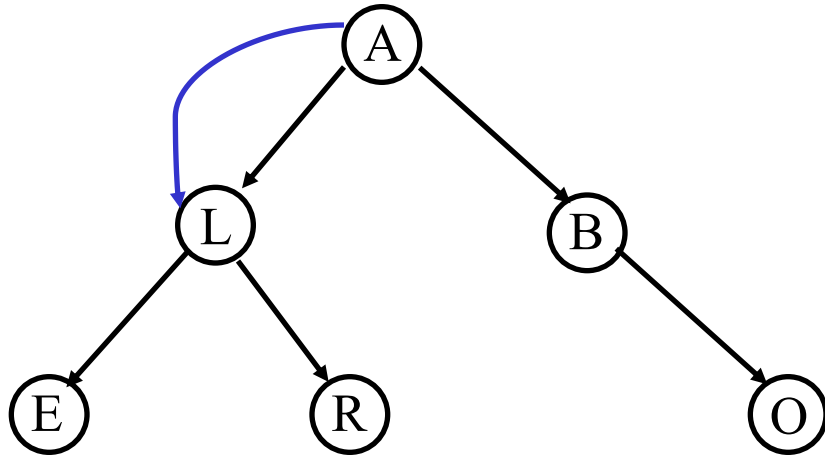
Ordine di visita: A





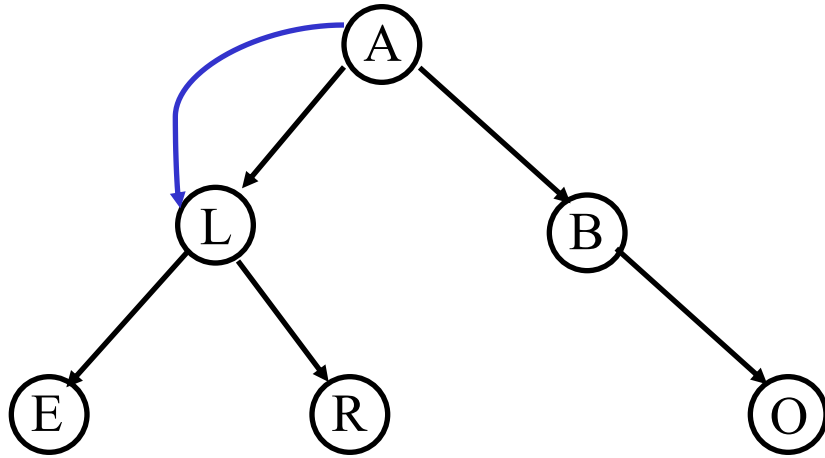
Ordine di visita: A

L  
B



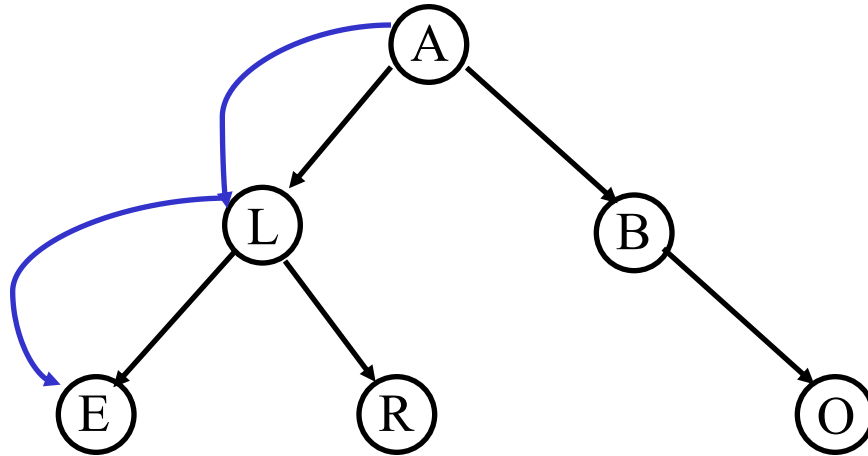
Ordine di visita: A L

B



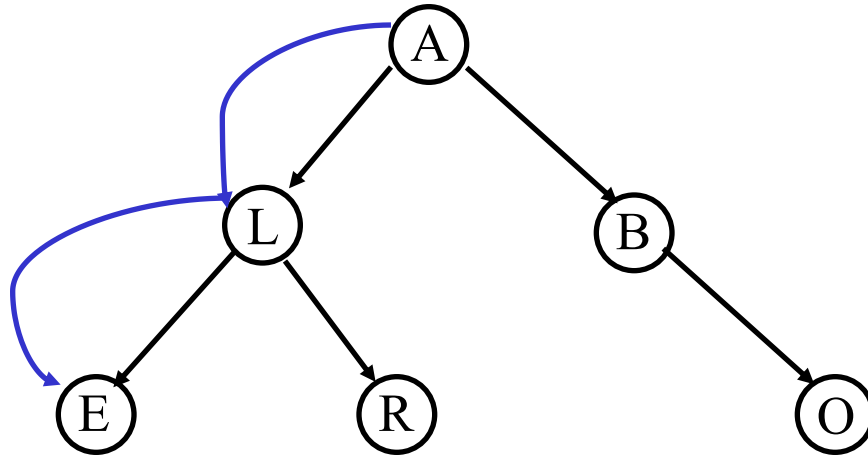
Ordine di visita: A L

E  
R  
B



Ordine di visita: A L E

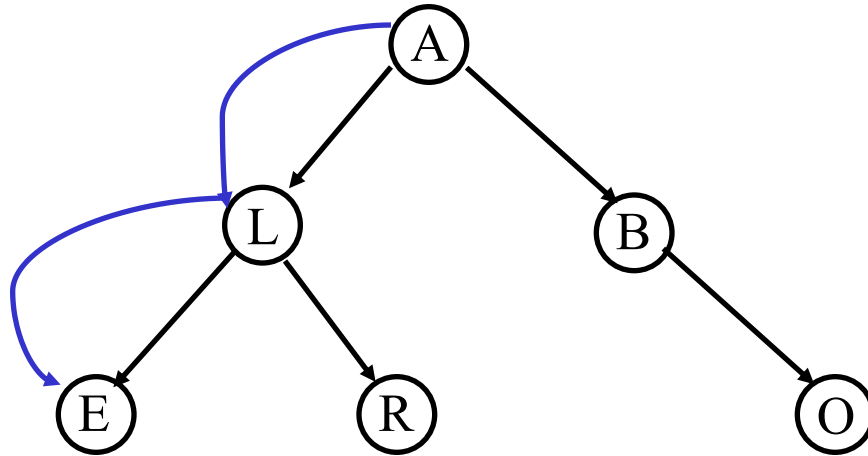
R  
B



Ordine di visita: A L E

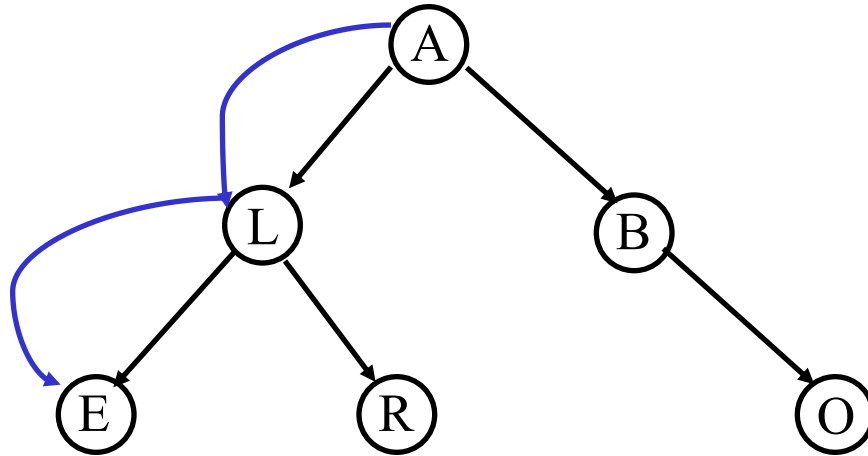
null
null
R
B





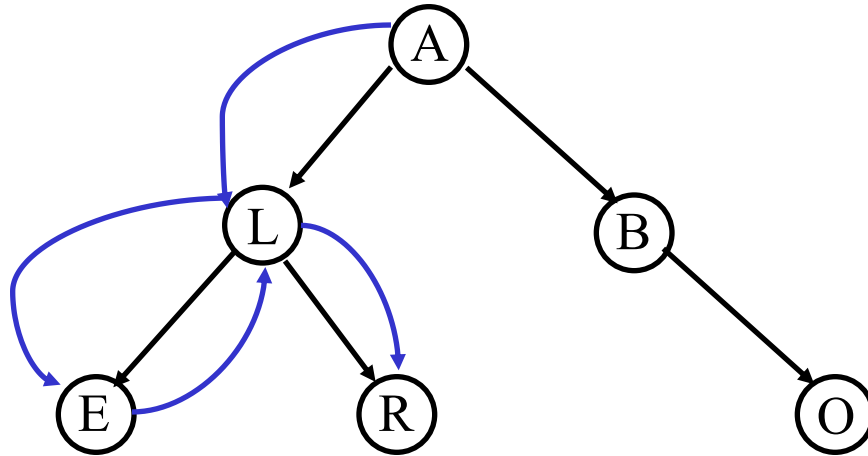
Ordine di visita: A L E

null  
R  
B



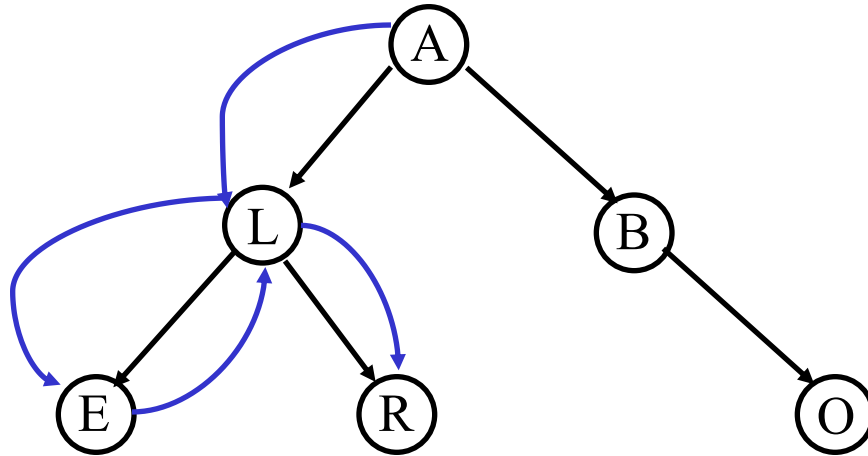
Ordine di visita: A L E

R  
B



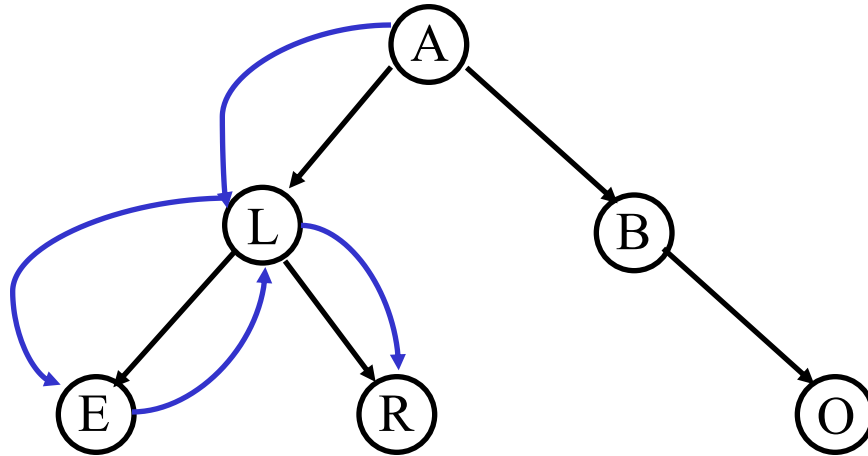
Ordine di visita: A L E R

B



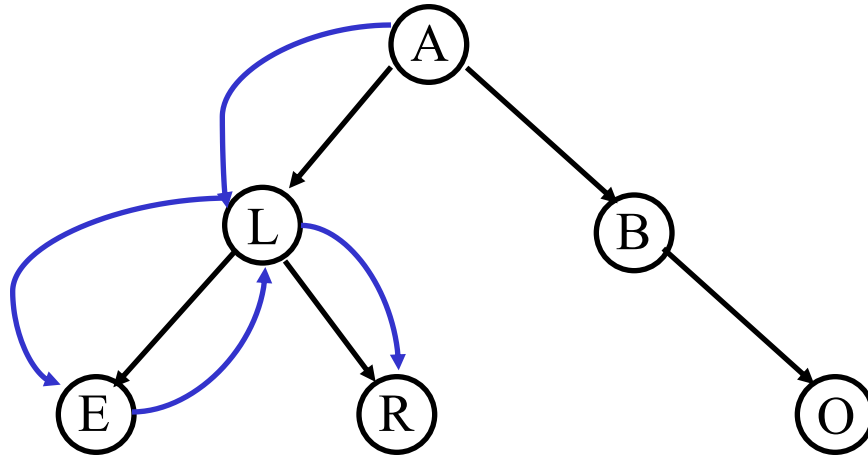
Ordine di visita: A L E R

null  
null  
B



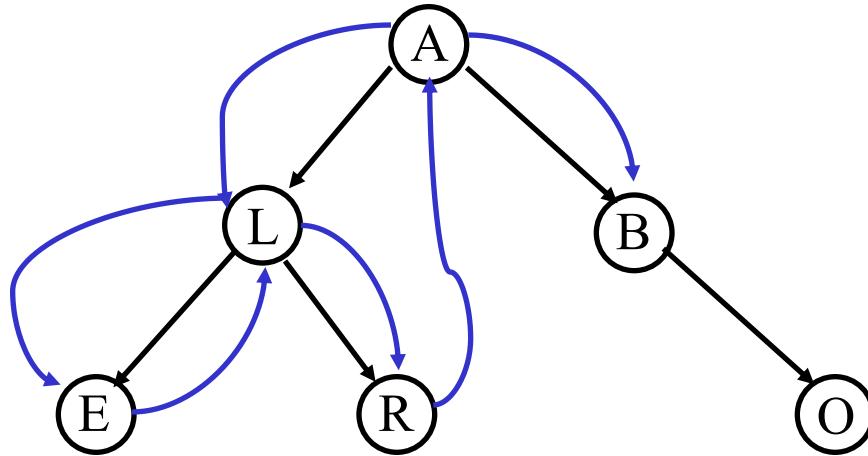
Ordine di visita: A L E R

null  
B



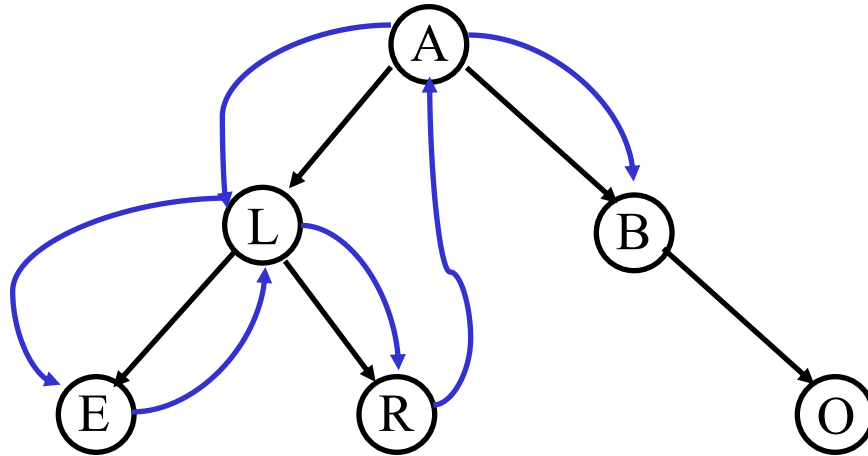
Ordine di visita: A L E R

B



Ordine di visita: A L E R B

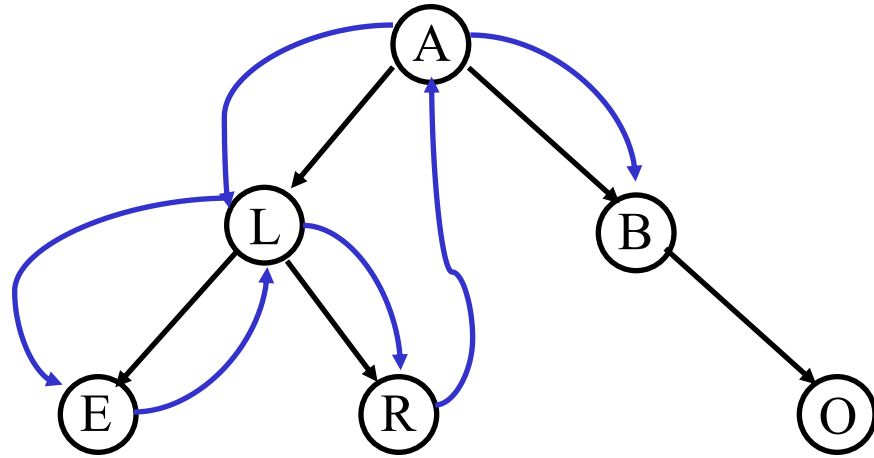




Ordine di visita: A L E R B

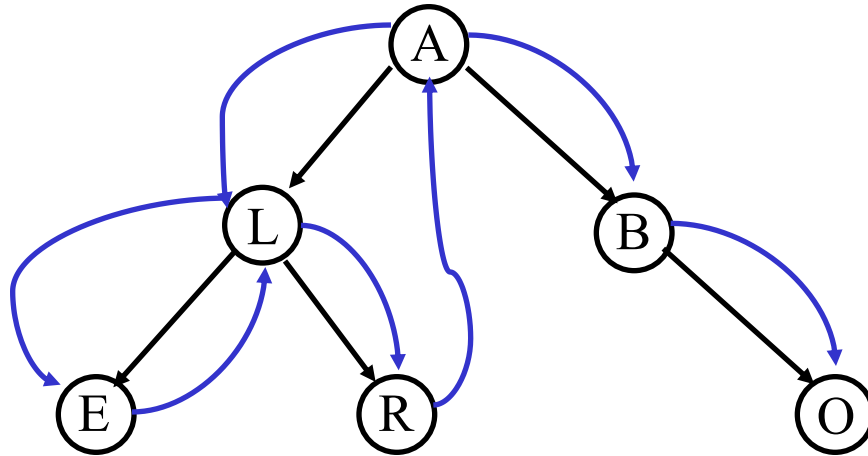
null
O





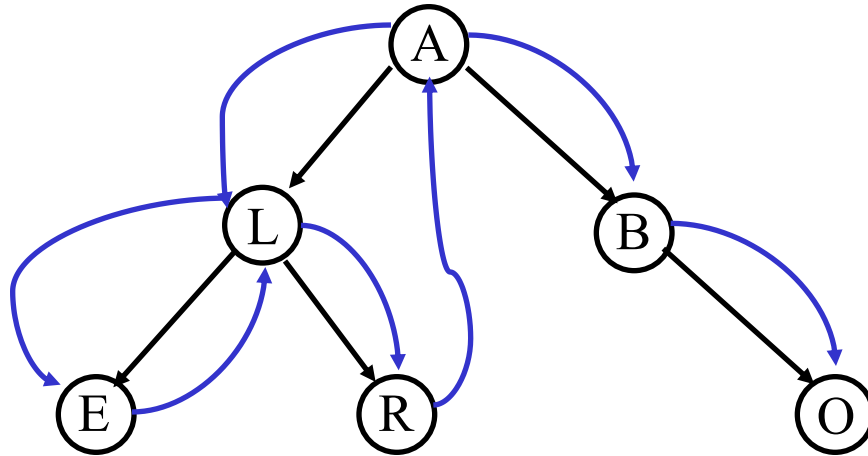
Ordine di visita: A L E R B

O



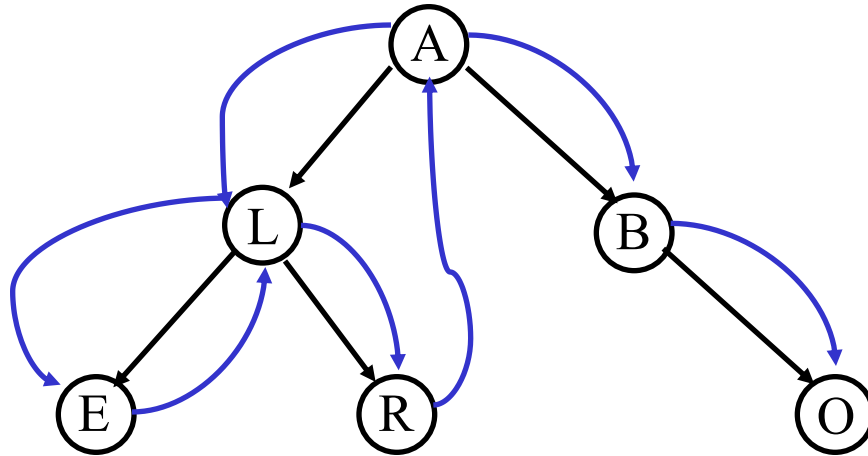
Ordine di visita: A L E R B O





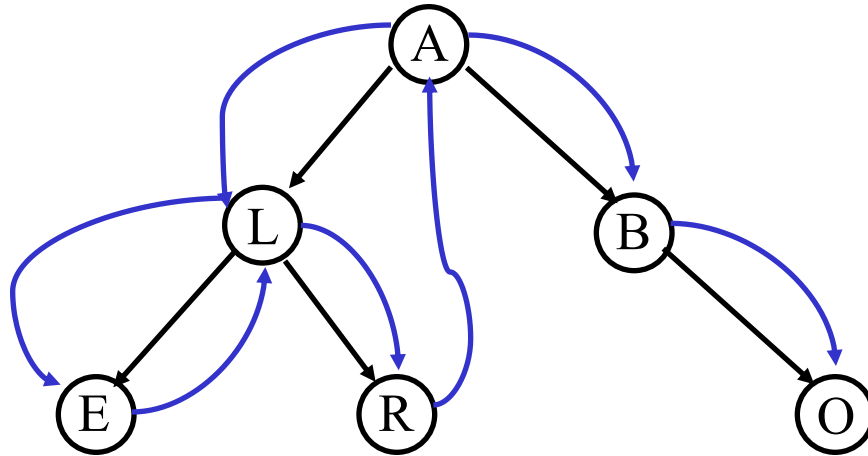
Ordine di visita: A L E R B O

null
null



Ordine di visita: A L E R B O

⌋  
null



Ordine di visita: A L E R B O



# Complessità temporale

```
algoritmo visitaDFS(nodo  $r$ )  
  Pila  $S$   
   $S.push(r)$   
  while (not  $S.isEmpty()$ ) do  
     $u \leftarrow S.pop()$   
    if ( $u \neq null$ ) then  
      visita il nodo  $u$   
       $S.push(\text{figlio destro di } u)$   
       $S.push(\text{figlio sinistro di } u)$ 
```

Ogni nodo inserito e estratto dalla Pila una sola volta

Tempo speso per ogni nodo:  $O(1)$   
(se so individuare i figli di un nodo in tempo costante)

# nodi null inseriti/estratti:  $O(n)$

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

# Algoritmo di visita in profondità

## Versione ricorsiva (per alberi binari):

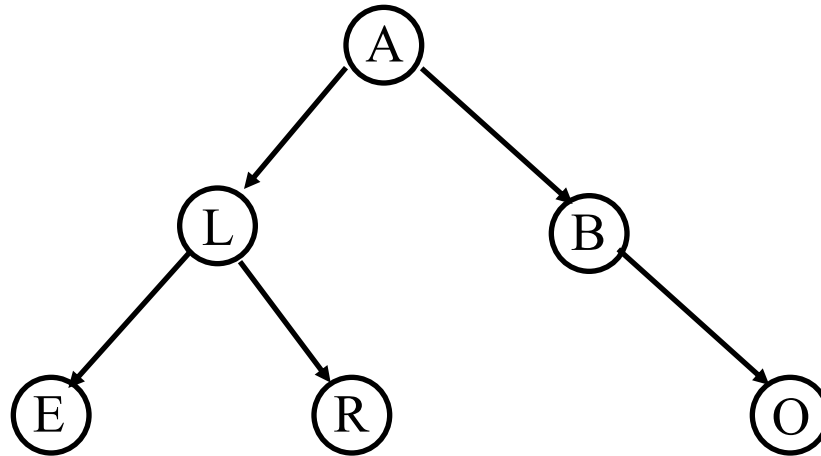
```
algoritmo visitaDFSRicorsiva(nodo r)  
1.   if ( $r \neq \text{null}$ ) then  
2.   visita il nodo r  
3.   visitaDFSRicorsiva(figlio sinistro di  $r$ )  
4.   visitaDFSRicorsiva(figlio destro di  $r$ )
```

**Visita in preordine:** radice, sottoalbero sin, sottoalbero destro

**Visita simmetrica:** sottoalbero sin, radice, sottoalbero destro  
(scambia riga 2 con 3)

**Visita in postordine:** sottoalbero sin, sottoalbero destro, radice  
(sposta riga 2 dopo 4)

...esempi...



Preordine: A L E R B O

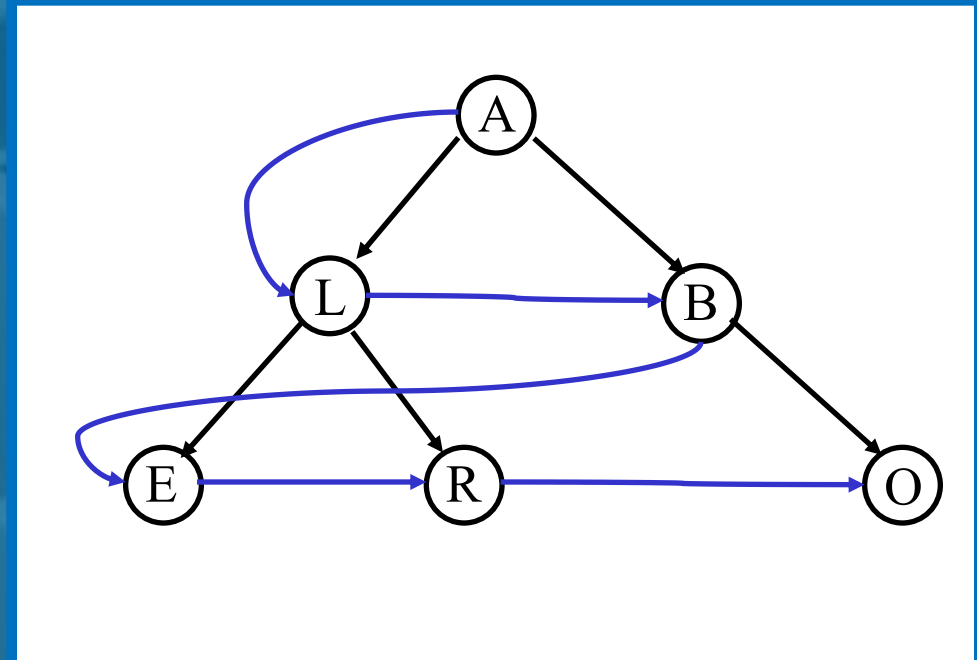
Simmetrica: E L R A B O

Postordine: E R L O B A



# Algoritmo di visita in ampiezza

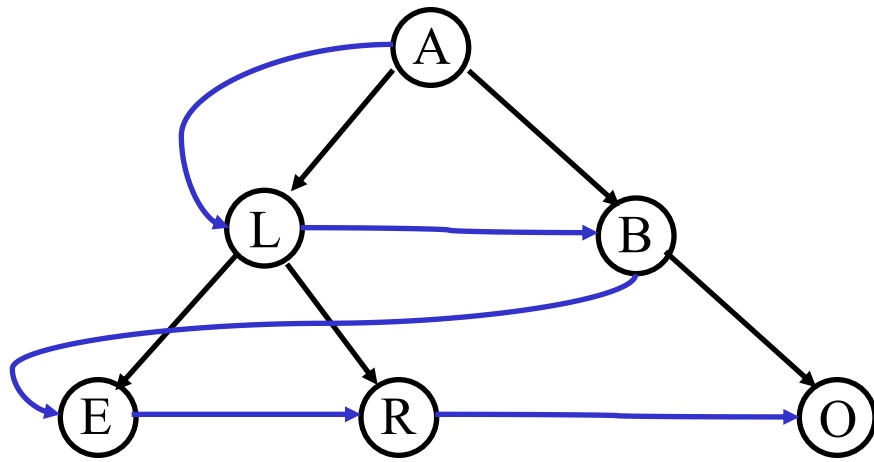
L'algoritmo di visita in ampiezza (BFS) parte da  $r$  e procede visitando nodi per livelli successivi. Un nodo sul livello  $i$  può essere visitato solo se tutti i nodi sul livello  $i-1$  sono stati visitati.



# Algoritmo di visita in ampiezza

Versione iterativa (per alberi binari):

```
algoritmo visitaBFS(nodo r)  
  Coda C  
  C.enqueue(r)  
  while (not C.isEmpty()) do  
    u ← C.dequeue()  
    if (u ≠ null) then  
      visita il nodo u  
      C.enqueue(figlio sinistro di u)  
      C.enqueue(figlio destro di u)
```



**Nota:**  
 inserisco nella  
 coda solo nodi  
 diversi da **null**

Ordine di visita: A L B E R O



# Complessità temporale

```
algoritmo visitaBFS(nodo r)  
  Coda C  
  C.enqueue(r)  
  while (not C.isEmpty()) do  
    u ← C.dequeue()  
    if (u ≠ null) then  
      visita il nodo u  
      C.enqueue(figlio sinistro di u)  
      C.enqueue(figlio destro di u)
```

Ogni nodo inserito e estratto dalla Pila una sola volta

Tempo speso per ogni nodo:  $O(1)$   
(se so individuare i figli di un nodo in tempo costante)

# nodi null  
inseriti/estratti:  $O(n)$

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

# Utilizzo algoritmi di visita un esempio: calcolo dell'altezza

CalcolaAltezza (nodo  $r$ )

1. **if** ( $r = \text{null}$ ) **then return** -1
2.  $sin = \text{CalcolaAltezza}(\text{figlio sinistro di } r)$
3.  $des = \text{CalcolaAltezza}(\text{figlio destro di } r)$
4. **return**  $1 + \max\{sin, des\}$

Calcola (e ritorna)  
l'altezza di  
un albero binario  
con radice  $r$

## Complessità temporale: $O(n)$

## Problema 3.6

Si scrivano varianti dell'algoritmo per:

1. calcolare il numero di foglie di un albero;
2. calcolare il grado medio dei nodi dell'albero (numero medio di figli di un nodo *non foglia*);
3. verificare se esiste un nodo dell'albero che abbia un dato contenuto informativo.

## Soluzione Problema 3.6.1

CalcolaNumFoglie (nodo  $r$ )

1. **if** ( $r = \text{null}$ ) **then return** 0
2. **if** ( $r$  è una foglia) **then return** 1
3.  $sin = \text{CalcolaNumFoglie}(\text{figlio sinistro di } r)$
4.  $des = \text{CalcolaNumFoglie}(\text{figlio destro di } r)$
5. **return** ( $sin + des$ )

Calcola il numero di  
foglie di un albero con  
radice  $r$

Complessità temporale:  $O(n)$

## Soluzione Problema 3.6.2

CalcolaGradoMedio (nodo  $r$ )

1.  $n$  = numero nodi dell'albero
2.  $nfoglie$  = CalcolaNumFoglie (nodo  $r$ )
3. **if** ( $r \neq \text{null}$ ) **return** (SommaGradi( $r$ )/( $n-nfoglie$ ))

Calcola grado medio  
dei nodi di un albero  
con radice  $r$

SommaGradi(nodo  $r$ )

1. **if** ( $r = \text{null}$ ) **return** 0
2. **if** ( $r$  è una foglia) **return** 0
3.  $S$  = numero figli di  $r$  + SommaGradi(figlio sinistro di  $r$ ) +  
SommaGradi(figlio destro di  $r$ )
4. **return**  $S$

Complessità temporale:  $O(n)$



## Soluzione Problema 3.6.3

CercaElemento (nodo  $r$ , chiave  $k$ )

1. **if** ( $r = \text{null}$ ) **then return** null
2. **if** ( $\text{chiave}(r) = k$ ) **then return**  $r$
3.  $\text{sin} = \text{CercaElemento}(\text{figlio sinistro di } r, k)$
4. **if** ( $\text{sin} \neq \text{null}$ ) **then return**  $\text{sin}$
5. **return**  $\text{CercaElemento}(\text{figlio destro di } r, k)$

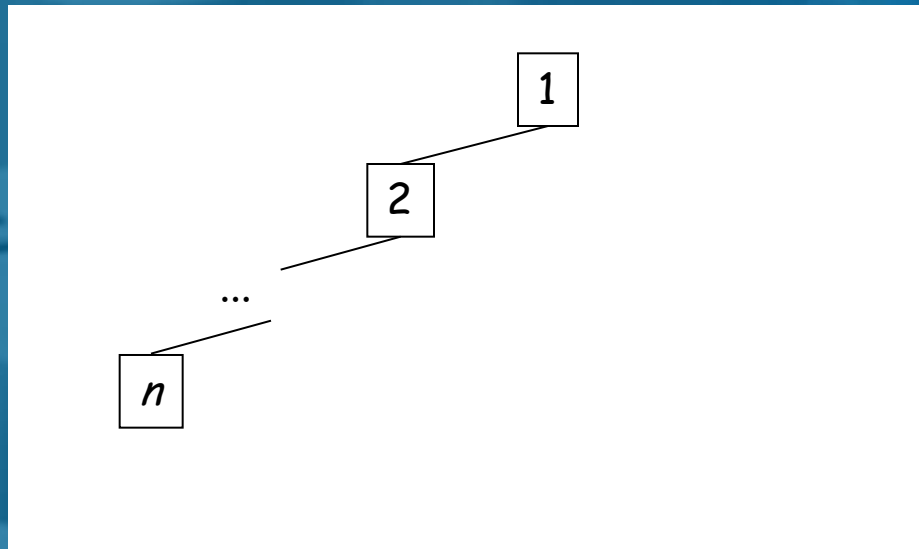
ritorna un nodo dell'albero di radice  $r$  che ha chiave  $k$ ; se tale nodo non esiste ritorna null

Complessità temporale:  $O(n)$

## Problema 3.3

Si consideri la rappresentazione di alberi basata su vettore posizionale. In principio, è possibile rappresentare in questo modo anche alberi non completi, semplicemente marcando come inutilizzate le celle che non corrispondono a nodi dell'albero. Quanto spazio potrebbe essere necessario per memorizzare un albero non completo con  $n$  nodi? Si assuma  $d=2$ .

## Soluzione Problema 3.3



Si consideri un albero di  $n$  nodi che è una *catena*, ovvero un albero tale che ogni nodo ha al più un figlio

L'altezza di questo albero è  $n-1$

L'albero binario completo di altezza  $n-1$  ha  $2^n-1$  nodi

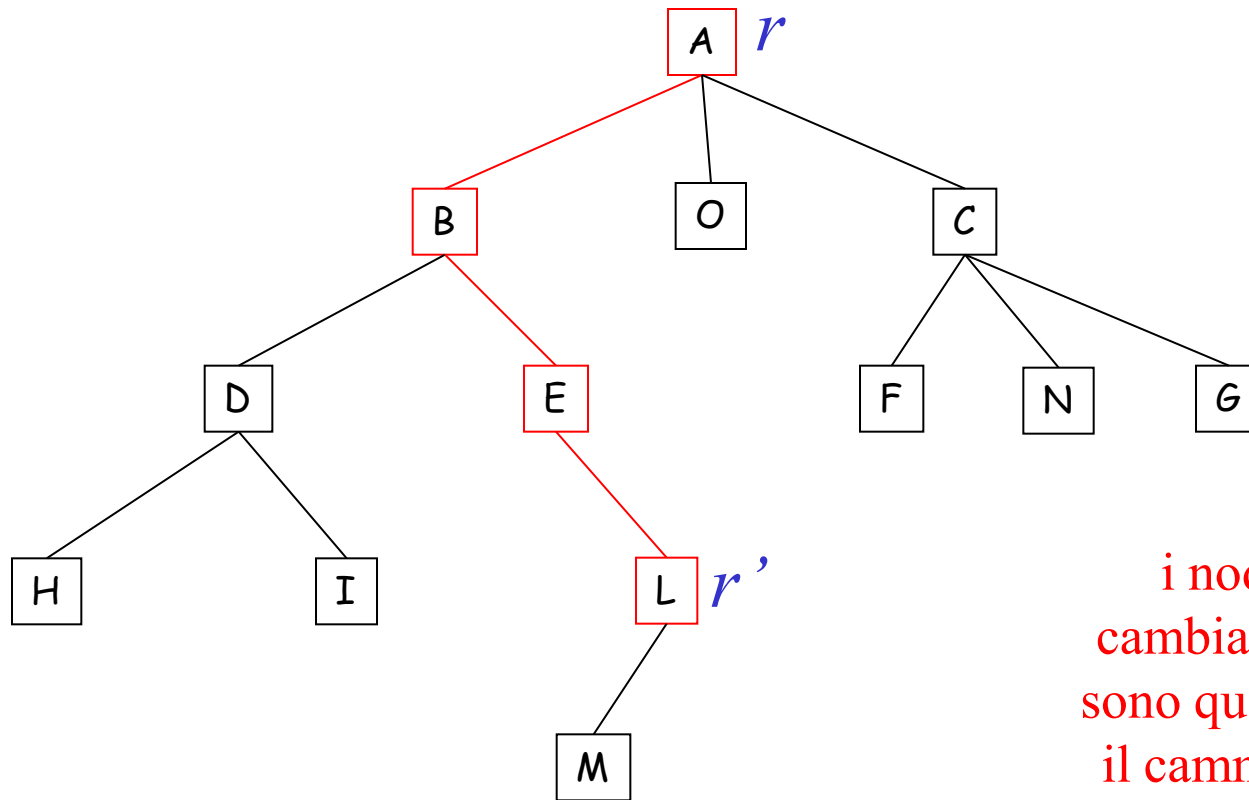
Quindi: dimensione vettore posizionale è  $2^n-1$

Quantità di memoria necessaria per memorizzare albero è **esponenzialmente più grande** del numero di nodi

# Esercizio

Sia  $T$  un albero (con radice  $r$ ) mantenuto attraverso un vettore dei padri. Progettare un algoritmo che, dato  $T$  e un nodo  $r'$  di  $T$ , restituisce il vettore dei padri che rappresenta  $T$  radicato in  $r'$ .

**Suggerimento:** quali sono i nodi che rispetto alla nuova radice cambiano padre?



i nodi che cambiano padre sono quelli lungo il cammino che unisce  $r'$  con  $r$

- ...il padre di L diventa null...
- ...il padre di E diventa L...
- ...il padre di B diventa E...
- ...il padre di A diventa B...

## Un possibile pseudocodice

RiRadica ( $T, j$ )

1.  $x=j$
2.  $px=T[j].parent$
3.  $T[j].parent= null$
4. **while** ( $px \neq null$ ) **do**
5.      $y=T[px].parent$
6.      $T[px].parent=x$
7.      $x=px$
8.      $px=y$
9. **endwhile**

Complessità temporale:  
 $O(h)$

dove  $h$  è l'altezza di  $T$   
rispetto alla radice  $r$