Alberi: TDD Tree, Visite, strutture dati per TDD Tree

Queste slide integrano parti del materiale a corredo del libro di testo “Algoritmi e Strutture Dati” di Camil Demetrescu, Irene Finocchi, Giuseppe F. Italiano (cap. 3, cap 6), rilasciato ai docenti dei corsi e protetto da copyright da parte della casa editrice McGraw-Hill. Vengono rese disponibili per ragioni didattiche agli studenti di ASD@Informatica-UniGE, che si impegnano a non rilasciarle ad altri e a non renderle pubbliche.

TDD Tree
(alcune possibili operazioni, dal libro di testo)

tipo `Albero`:
dati:
un insieme di nodi (di tipo *nodo*) e un insieme di archi.

operazioni:

`numNodi`() → *intero*
restituisce il numero di nodi presenti nell'albero.

`grado`(*nodo* v) → *intero*
restituisce il numero di figli del nodo v.

`padre`(*nodo* v) → *nodo*
restituisce il padre del nodo v nell'albero, o `null` se v è la radice.

`figli`(*nodo* v) → ⟨*nodo*, *nodo*, . . . , *nodo*⟩
restituisce, uno dopo l'altro, i figli del

`aggiungiNodo`(*nodo* u) → *nodo*
inserisce un nuovo nodo v come figlio di u nell'albero e lo restituisce. Se v è il primo nodo ad essere inserito nell'albero, esso diventa la radice (e u viene ignorato).

Errore nel testo: dovrebbe essere
aggiungiNodo(*nodo u, nodo v*) -> *nodo*

`aggiungiSottoalbero`(`Albero` a, *nodo* u)
inserisce nell'albero il sottoalbero a in modo che la radice di a diventi figlia di u.

`rimuoviSottoalbero`(*nodo* v) → `Albero`
stacca e restituisce l'intero sottoalbero radicato in v. L'operazione cancella dall'albero il nodo v e tutti i suoi discendenti.

TDD Tree

- Approccio del libro di testo: definire le operazioni direttamente sull'insieme dei nodi.
- Altro approccio possibile: assumere che i nodi dell'albero siano etichettati con etichette in un insieme Label e che ogni nodo abbia etichetta unica. Definire quindi le operazioni sul TDD Tree in termini delle etichette, viste come identificatori univoci dei nodi.
- **Svantaggi dell'uso diretto dei nodi**: mentre l'etichetta è un concetto astratto che prescinde dall'implementazione (nella pratica sarà rappresentata da una stringa, un intero, un URI, ma l'interfaccia delle operazioni non cambia se cambia tale rappresentazione), il concetto di “nodo” è molto legato a quale struttura dati viene usata per implementare il TDD Tree (potrà essere un indice in un array, un puntatore a una struct, ...) quindi nella pratica si perde un livello di astrazione e si espongono dettagli sulla struttura dati, che sarebbe molto più opportuno nascondere, nel rispetto dei principi dell'information hiding e dell'incapsulazione
- **Vantaggi**: proprio perché il “nodo” è qualcosa di molto vicino all'implementazione, definire le operazioni sul TDD Tree direttamente in termini dei nodi dell'albero permette solitamente implementazioni delle operazioni più efficienti, rispetto all'uso delle etichette che introducono un livello intermedio rispetto all'implementazione

TDD Tree

- Un ragionevole compromesso: definire il TDD in termini di etichette ma impiegare funzioni ausiliarie che operino direttamente sui nodi, in modo efficiente.
- Solitamente l'efficienza dell'implementazione delle operazioni di un TDD va a scapito della leggibilità e portabilità del codice. Bisogna trovare il giusto equilibrio.

TDD Tree

(versione “nostra”, con etichette come argomenti)

Sort coinvolte nell'algebra degli alberi: Tree, Label, Bool, Int, List

emptyTree: -> Tree

/* emptyTree è la costante corrispondente all'albero vuoto */

emptyLabel: -> Label

/* emptyLabel è la costante corrispondente all'etichetta vuota */

isEmpty: Tree -> Bool

/* isEmpty(t) restituisce true se l'albero t è vuoto, false altrimenti */

TDD Tree

(versione “nostra”, con etichette come argomenti)

addElem: Label x Label x Tree -> Tree

/* addElem(labelOfNodeInTree, labelOfNodeToAdd, t) aggiunge il nodo etichettato con labelOfNodeToAdd come figlio del nodo etichettato con labelOfNodeInTree.

Caso particolare: aggiunta della radice, che si ottiene con labelOfNodeInTree uguale a emptyLabel (= nessun padre) e ha successo solo se l'albero t è vuoto. Se non esiste un nodo nell'albero etichettato con labelOfNodeInTree oppure se nell'albero esiste gia' un nodo etichettato con labelOfNodeToAdd non aggiunge nulla */

TDD Tree

(versione “nostra”, con etichette come argomenti)

deleteElem: Label x Tree -> Tree

/* deleteElem(l, t) rimuove dall'albero il nodo etichettato con la Label l e collega al padre di tale nodo tutti i suoi figli. Fallisce se si tenta di cancellare la radice e questa ha dei figli (non si saprebbe a che padre attaccarli) oppure se non esiste un nodo nell'albero etichettato con l */

father: Label x Tree -> Label

/* father(l, t) restituisce l'etichetta del padre del nodo etichettato l se il nodo esiste nell'albero (o sottoalbero) t e se il padre esiste. Restituisce la costante emptyLabel altrimenti */

TDD Tree
(versione “nostra”, con etichette come argomenti)

children: Label x Tree -> List<Label>

/* children(l, t) restituisce una lista di Label contenente le etichette di tutti i figli del nodo etichettato con l nell'albero t */

degree: Label x Tree -> Int

/* degree(l, t) restituisce il numero dei figli del nodo etichettato con l se il nodo etichettato con l esiste; restituisce qualche valore concordato (ad es. -1) altrimenti */

ancestors: Label x Tree -> List<Label>

/* ancestors(l, t) restituisce una lista di Label contenente le etichette di tutti gli antenati del nodo l ESCLUSA l'etichetta del nodo stesso */

TDD Tree

(versione “nostra”, con etichette come argomenti)

leastCommonAncestor: Label x Label x Tree -> Label

/* leastCommonAncestor(label1, label2, t) restituisce l'etichetta del minimo antenato comune ai nodi etichettati con label1 e label2 nell'albero t */

member: Label x Tree -> Bool

/* member(l, t) restituisce true se il nodo etichettato con la label l appartiene all'albero t e false altrimenti */

numNodes: Tree -> Int

/* numNodes(t) restituisce il numero di nodi in t */

Visite di alberi

Visite di alberi (da testo DFI)

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

Visita generica (da testo DFI)

visitaGenerica visita il nodo r e tutti i suoi discendenti in un albero. S è un insieme.

algoritmo `visitaGenerica`$(nodo\ r)$

1. $S \leftarrow \{r\}$
2. **while** $(S \neq \emptyset)$ **do**
3. estrai un nodo u da S
4. *visita il nodo u*
5. $S \leftarrow S \cup \{$ figli di $u\ \}$

(“estrai” = “seleziona e rimuovi da S”, altrimenti non funziona!)

Richiede tempo Θ(n) per visitare un albero con n nodi a partire dalla radice

Visita in profondità (DFS)

- La visita in profondità (depth-first search, DFS), intrinsecamente ricorsiva, scende in profondità nella struttura dell'albero.
- L’algoritmo di visita in profondità parte dalla radice r e procede visitando i nodi di figlio in figlio fino a raggiungere una foglia. Retrocede (“*backtrack*”) poi al primo antenato che ha ancora figli non visitati (se esiste) e ripete il procedimento a partire da uno di quei figli.

Visita DFS per alberi binari
Versione iterativa

```
algoritmo visitaDFS(nodo r)
    Pila S
    S.push(r)
    while (not S.isEmpty()) do
        u ←S.pop()
        if (u ≠ null) then
            visita il nodo u
            S.push(figlio destro di u)
            S.push(figlio sinistro di u)
```

Si generalizza banalmente al caso di un albero generico, facendo la push di tutti i figli del nodo u dopo aver visitato u:
while(u ha ancora un figlio f)
  push(f, S)

Visita DFS per alberi binari
Versione ricorsiva

algoritmo `visitaDFSRicorsiva`(*nodo* r)

1. **if** ($r =$ `null`) **then return**
2. *visita il nodo* r
3. `visitaDFSRicorsiva`(figlio sinistro di r)
4. `visitaDFSRicorsiva`(figlio destro di r)

Si generalizza banalmente al caso di un albero generico, chiamando la visitaDFSRicorsiva su ogni figlio del nodo r
while(r ha ancora un figlio f)
  visitaDFSRicorsiva(f)

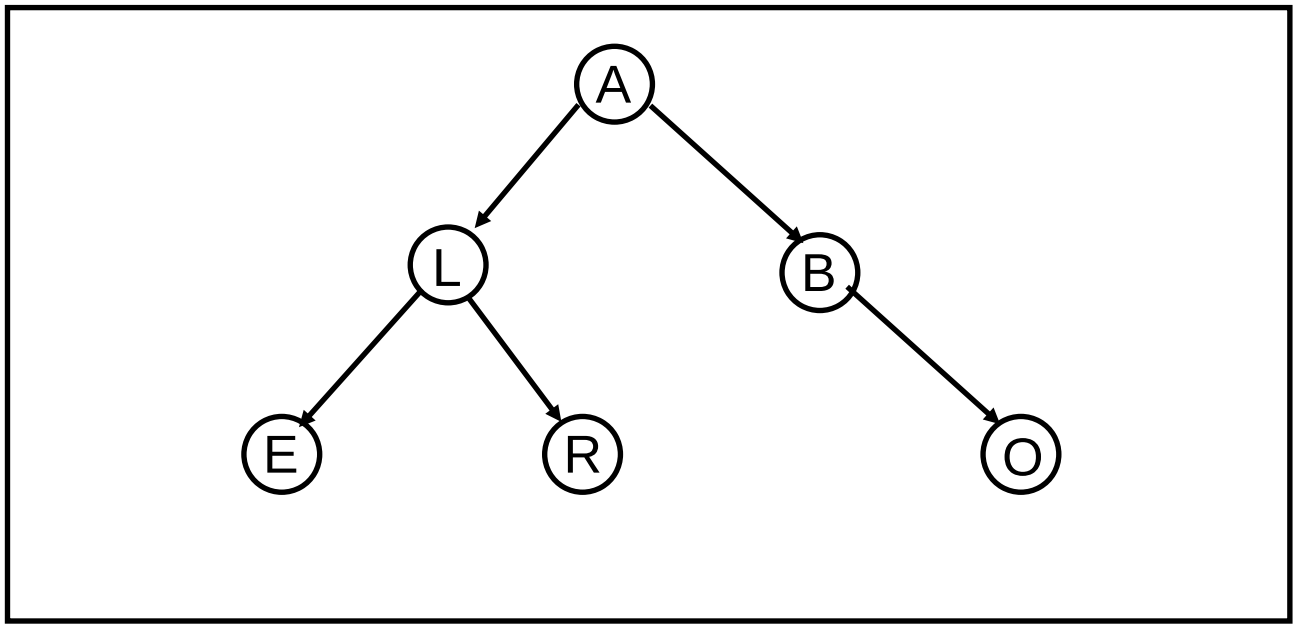
Visite preordine, simmetrica, postordine di alberi binari

Visita in preordine: radice, sottoalbero sin, sottoalbero destro

Visita simmetrica: sottoalbero sin, radice, sottoalbero destro (scambia riga 2 con 3 nello pseudo-codice della DFS ricorsiva)

Visita in postordine: sottoalbero sin, sottoalbero destro, radice (sposta riga 2 dopo 4 nello pseudo-codice della DFS ricorsiva)

Esempi



Preordine: A L E R B O
Simmetrica: E L R A B O
Postordine: E R L O B A

Visita in ampiezza (BFS)

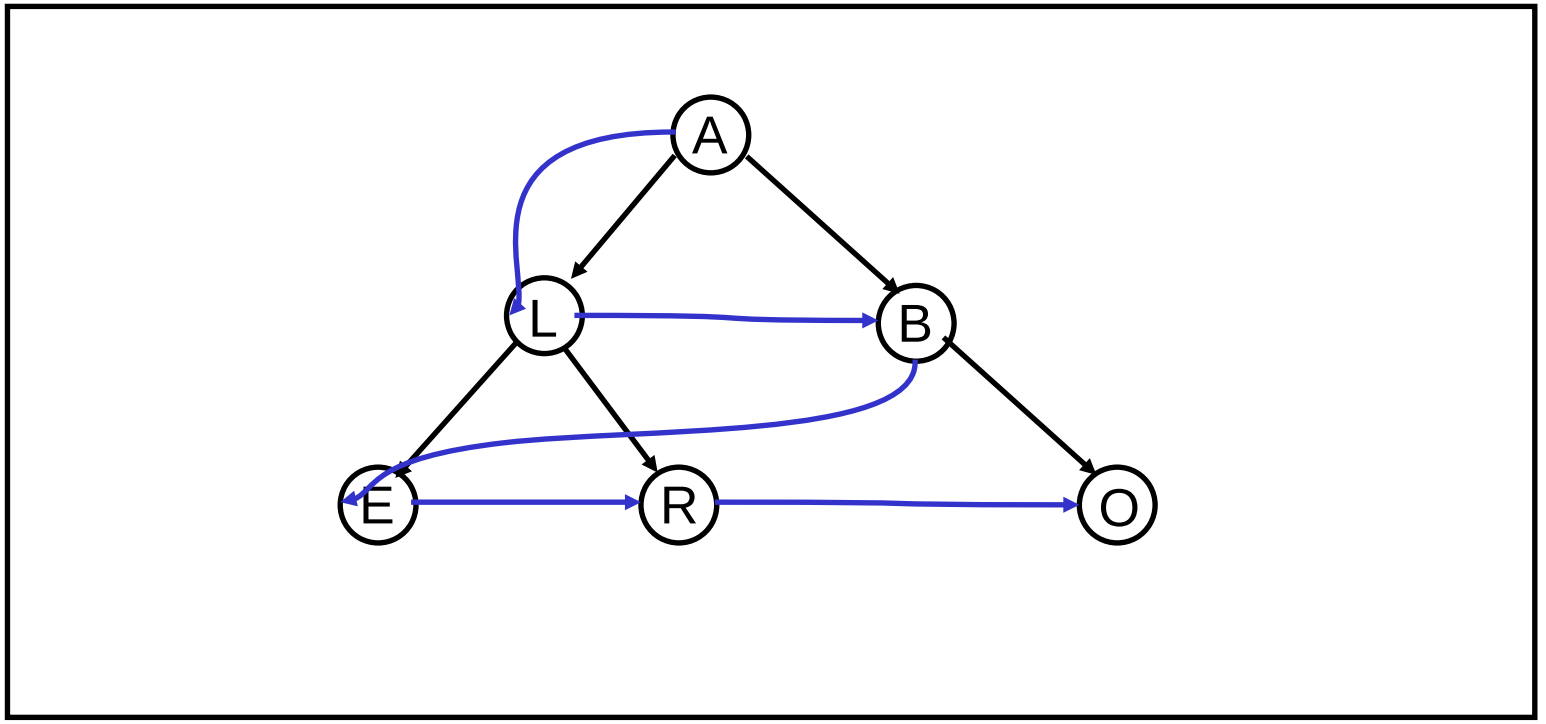
L’algoritmo di visita in ampiezza (breadth-first search, BFS) parte da r e procede visitando nodi per livelli successivi. Un nodo sul livello i può essere visitato solo se tutti i nodi sul livello i-1 sono stati visitati.

Visita BFS per alberi binari
Versione iterativa

```
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)
```

Si generalizza banalmente al caso di un albero generico, facendo la enqueue di tutti i figli del nodo u dopo averlo visitato
while(u ha ancora un figlio f)
  enqueue(f, C)

Esempi

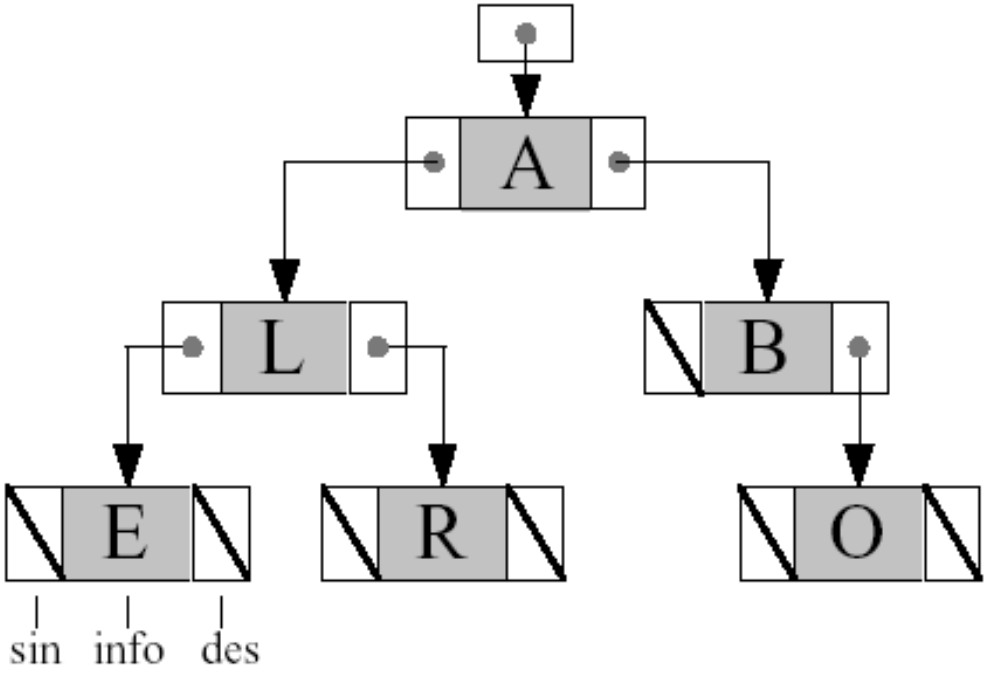


Visita BFS: A L B E R O

Come rappresentare alberi generici

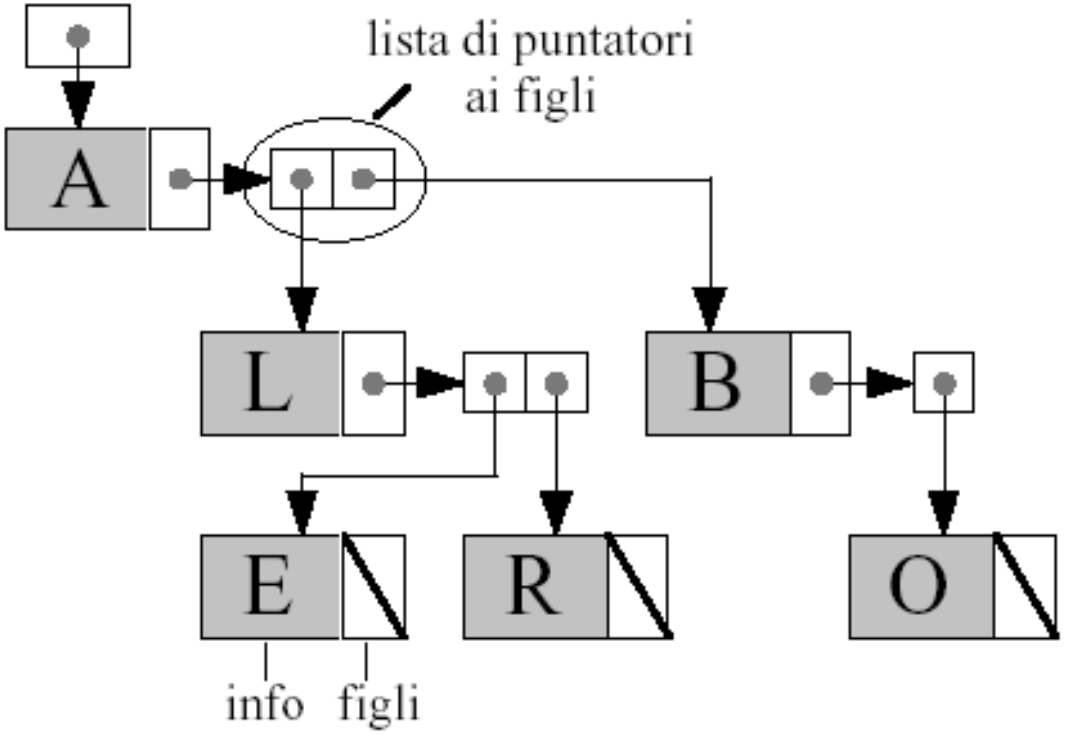
Rappresentazioni collegate di alberi generici

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



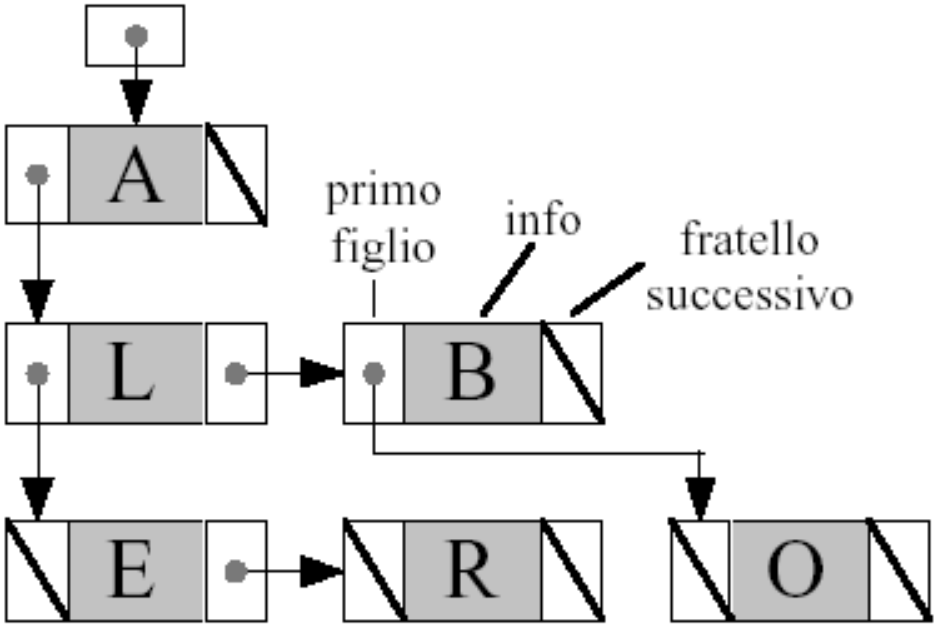
Rappresentazioni collegate di alberi generici

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



Rappresentazioni collegate di alberi generici

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



Rappresentazioni indicizzate di alberi generici

Idea: ogni cella dell’array contiene le informazioni di un nodo + eventualmente altri indici per raggiungere altri nodi

Rappresentazioni indicizzate di alberi d-ari quasi completi

Vettore posizionale per alberi d-ari quasi completi, ovvero quelli in cui possono mancare delle foglie consecutive a partire dalla prima foglia a destra:
- nodi disposti nell'array "per livelli", nello stesso ordine in cui verrebbero incontrati con una visita in ampiezza, BFS
- *j*-esimo figlio (*j* ϵ {1,…,*d*}) di *i* è in posizione *d**(*i*-1)+*j*+1 **[dimostrato per induzione sul livello dell'albero]**
- il padre di i (con i diverso da 1 che indica la radice) è in posizione *floor((i-2)/d) + 1* **[non dimostrato]**

Per comodità nei calcoli, non si usa la cella 0 dell'array

floor(x) è il più grande intero minore od uguale a x. Per esempio floor(2.9)=2, floor(−2) = −2 e floor(−2.3) = −3. La funzione floor ("parte intera") è anche indicata con int(x) o [x].

Questa rappresentazione funziona **solo** se l'albero è d-ario ed è quasi completo

Dimostrazione delle formule (per induzione sul livello dell'albero)

Proprietà: Il *j*-esimo figlio ($j \in \{1,\ldots,d\}$) di un nodo *i* che si trova **al livello n** è in posizione $d*(i-1)+j+1$

- **Base**: n=0

  Sia i l'indice del nodo al livello 0: i è l'indice della radice ed è, per come abbiamo costruito l'array, in posizione 1. I d figli della radice si troveranno nelle posizioni da 2 a 2+d-1 compresi. In particolare, il primo figlio della radice si trova in posizione 2, il secondo in posizione 3, il j-esimo in posizione 1+j = d*(i-1)+j+1, il d-esimo in posizione 2+d-1=d*(i-1)+d+1 [cvd]

Dimostrazione delle formule (per induzione sul livello dell'albero)

Proprietà: Il j-esimo figlio ($j \in \{1,\ldots,d\}$) di un nodo i che si trova **al livello n** è in posizione $d*(i-1)+j+1$

- **Passo induttivo**: se la proprietà vale per i nodi al livello n>0, allora vale per i nodi al livello n+1

  Sia i l'indice di un nodo al livello n>0: poiché l'albero è d-ario e quasi completo, al livello n si trovano n^d nodi. Il primo nodo del livello n>0 ha indice $d^{(n-1)} + 1$ e l'ultimo ha indice $d^{(n-1)} + d^n$.
- per l'ipotesi induttiva, il figlio j-esimo del nodo indicizzato con i si trova in posizione $p = d*(i-1)+j+1$.
- Devo dimostrare che il k-esimo figlio del nodo in posizione p si trova in posizione $d*(p-1)+k+1 = d*((d*(i-1)+j+1)-1) + k + 1 = d^2*(i-1) + d*j+ k+1$

Dimostrazione delle formule (per induzione sul livello dell'albero)

Proprietà: Il *j*-esimo figlio (*j* ϵ {1,…,*d*}) di un nodo *i* che si trova **al livello n** è in posizione $d*(i-1)+j+1$

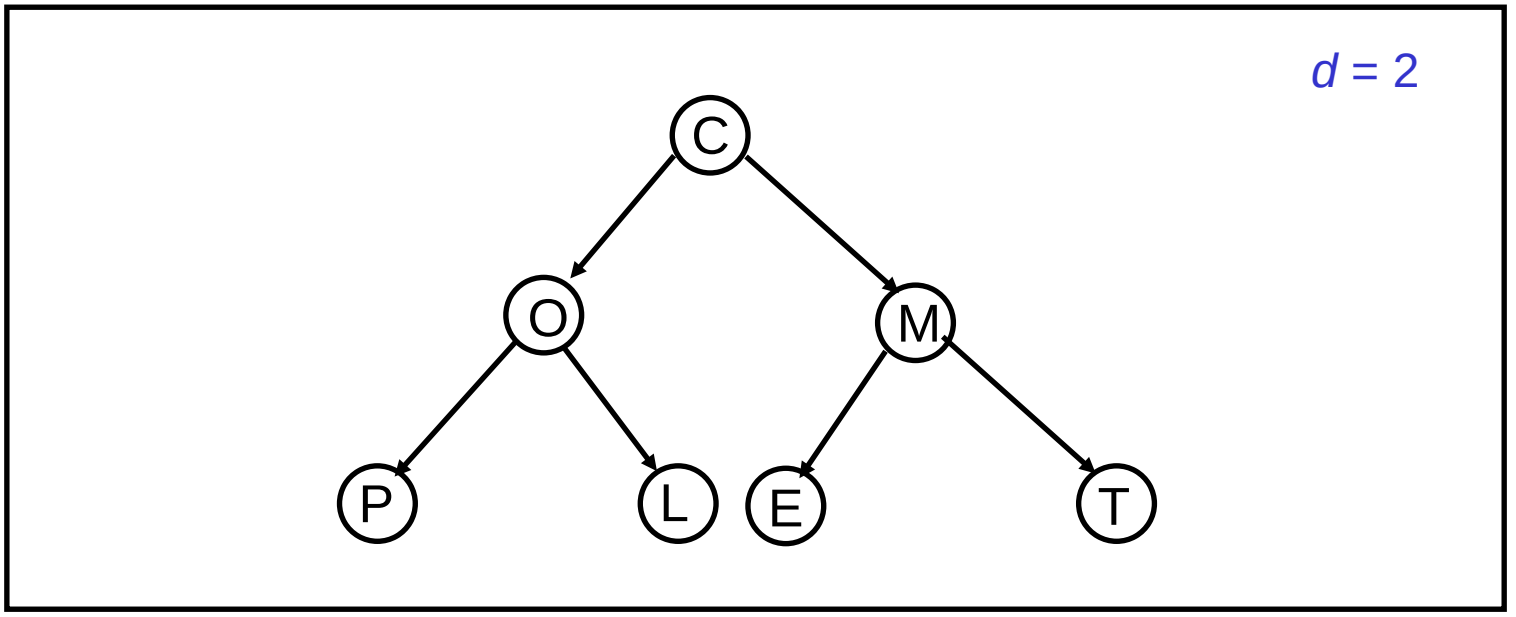
- **Passo induttivo**: se il nodo j-esimo di i si trova al livello n, il suo k-esimo figlio di trova al livello n+1; avrà quindi indice uguale a d^n*+1 (indice del primo nodo al livello n+1) + "il numero di nodi al livello n che vengono prima del figlio j-esimo di i"*d + k*
- "il numero di nodi al livello n che vengono prima del figlio j-esimo di i" è uguale all'indice del figlio j-esimo di i (che per ipotesi induttiva vale $d*(i-1)+j+1$) meno l'indice del primo nodo al livello n-esimo (che per come sono strutturati l'albero e l'array vale $d^{(n-1)} + 1$), quindi:

  $d*(i-1)+j+1 \; - \; (d^{(n-1)} + 1) = d*(i-1)+j-d^{(n-1)}$

  Questo valore va ancora moltiplicato per d, ottenendo $d^2*(i-1)+dj-d^n$
- La posizione del figlio k-esimo del figlio j-esimo di i è quindi:

  $d^n +1 + d^2*(i-1) + d*j -d^n + k = d^2*(i-1) + d*j+ k+1$ [cvd]

Rappresentazioni indicizzate di alberi d-ari quasi completi



A

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

Rappresentazioni indicizzate di alberi generici

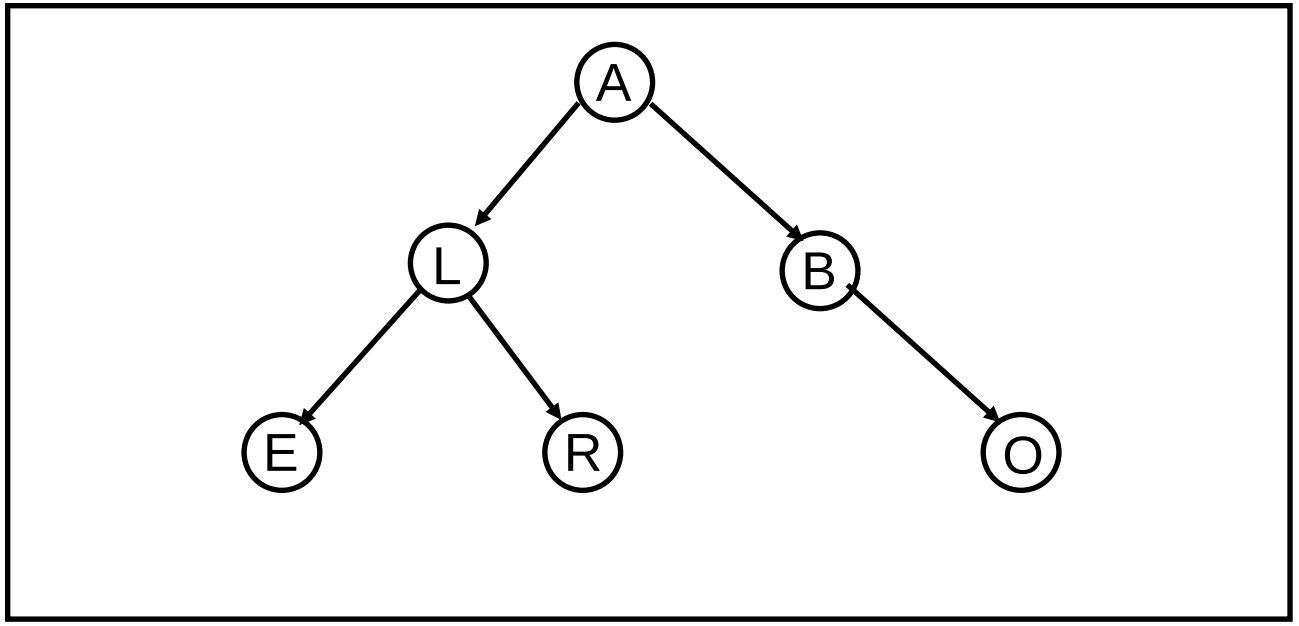
Vettore dei padri
Per un albero con n nodi uso un array P di dimensione almeno n

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

info: contenuto informativo del nodo *i*

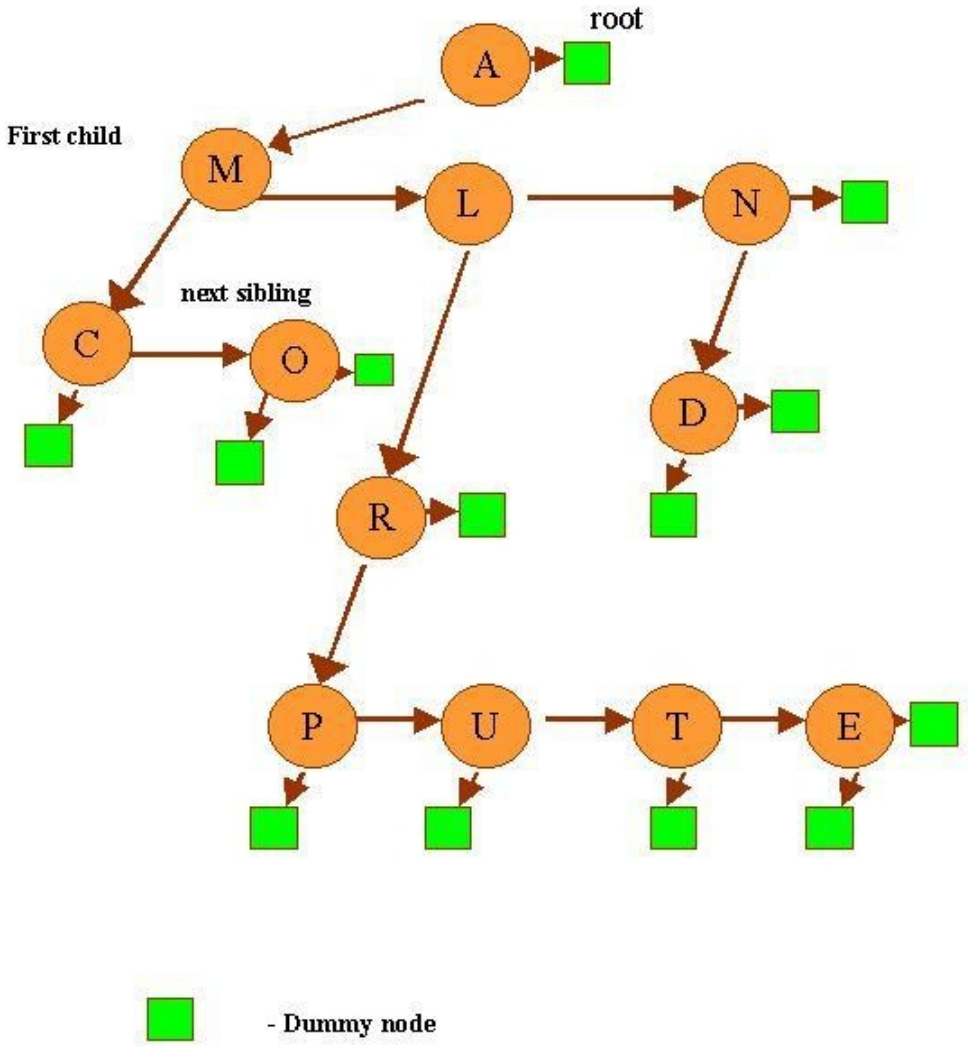
parent: indice (nell'array) del nodo padre di *i*

Rappresentazioni indicizzate di alberi generici



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

Spunti per l'implementazione primo figlio-prossimo fratello



Legenda

- Lo scopo delle slide seguenti è offrire una guida per una possibile implementazione delle operazioni del TDD Tree. La guida offerta è abbastanza vicina al codice da rappresentare un'utile traccia per l'implementazione, ma senza arrivare alla stesura di tutto il codice necessario. La traccia offerta non è l'unica possibile: è una delle tante.
- I commenti in rosso e italico stanno al posto di parti di codice da implementare, la cui implementazione richiede grande attenzione e cautela, ma non particolare fantasia ed inventiva.
- I commenti in nero spiegano parti di codice già implementate.

treeNode e Tree

```
struct tree::treeNode {
    Label label;
    treeNode *firstChild;
    treeNode *nextSibling;
};


typedef treeNode* Tree;
/* un albero è identificato dal puntatore
alla sua radice; useremo indistintamente
"albero" e "puntatore a nodo" */
```

Struttura della visita DFS in preordine

```
void visit(const Tree& t)
{
/* se l'albero e' vuoto non c'é niente da visitare */
    if (isEmpty(t)) return;


/* ...dovrò fare qualche operazione op(t) su t: potrebbe essere la stampa
dell'etichetta di t, il confronto dell'etichetta di t con un'etichetta che
stiamo cercando, la verifica di qualche altra proprietà di t o dei suoi
figli, etc... . E' l'operazione che nel testo viene indicata con “visita
di t”... */


/* Chiamata ricorsiva di visit su ciascuno dei figli di t, partendo dal
primo figlio e poi scorrendo la lista dei suoi fratelli */
    Tree auxT = t->firstChild;
    while (!isEmpty(auxT)) {
       visit(auxT);
       auxT=auxT->nextSibling;
    }
}
```

funzione ausiliaria getNode
(non è obbligatoria, ma può essere molto utile...)

```
/* restituisce il puntatore al nodo che si trova in t ed ha come
etichetta l */
Tree getNode(const Label l, const Tree& t)
{
    /* se t è vuoto o l'etichetta è vuota restituisco emptyTree */
    if (isEmpty(t) || l==emptyLabel)
        return emptyTree;

    /* se l'etichetta di t è quella cercata, restituisco t */
    if (t->label == l)
        return t;

    /* ...chiamata ricorsiva di getNode su ciascuno dei figli di t,
finché una delle chiamate non restituisce un valore diverso da
emptyTree.... Rispetto alla visita esaustiva vista prima, la getNode non
deve per forza esplorare tutto l'albero: quando una delle chiamate
restituisce un albero != emptyTree, si deve fare return di tale albero
non vuoto interrompendo la scansione dei fratelli */
}
```

addElem

```
Error tree::addElem(const Label labelOfNodeInTree, const Label
labelOfNodeToAdd, Tree& t)
{
   /* ...controlli iniziali: L'albero t e' vuoto? L'etichetta labelOfNodeInTree
e' emptyLabel? L'etichetta labelOfNodeToAdd è già presente nell'albero?.... */


   Tree auxT = getNode(labelOfNodeInTree, t); /* recupero il puntatore al nodo
dell'albero che ha etichetta labelOfNodeInTree */


   if (auxT == emptyTree)
     return FAIL; /* nell'albero non esiste un nodo con etichetta
labelOfNodeInTree, restituisco FAIL */
   else /* ho trovato il nodo auxT a cui aggiungere il figlio */
     {
     /* creo il figlio con etichetta labelOfNodeToAdd e lo aggiungo a auxT. Può
essere una buona idea implementare una funzione ausiliaria che crea il nodo e
ne restituisce il puntatore */
     }
   return OK;
}
```

deleteElemI
(versione iterativa)

```
Error tree::deleteElemI(const Label l, Tree& t)

{

    /* ...controlli iniziali: il nodo da rimuovere c'é?*/

     Tree fatherTree = getNode(father(l,t), t); /* recupero il puntatore al padre del nodo da rimuovere */

     if (isEmpty(fatherTree))  /* se fatherTree e' vuoto, sto tentando di rimuovere la radice */

           /* ....gestione del caso in cui si tenta di rimuovere la radice (la rimozione è possibile solo se la radice non ha figli; t deve diventare l'albero vuoto) */

     else /* sto tentando di rimuovere un nodo interno */

           {

           Tree nodeToRemove = getNode(l, t);   /* recupero il puntatore al nodo da rimuovere */

           /* ...i figli del nodo da rimuovere devono diventare suoi fratelli...*/

           /* ...poi devo rimuovere il nodo: se è il “firstChild” di suo padre, il puntatore del padre al firstChild deve cambiare, puntando al figlio successivo; altrimenti il nodo si trova in mezzo alla lista dei sibling del firstChild: bisogna rimuoverlo aggiustando i puntatori... */

           delete nodeToRemove; /* in ogni caso, alla fine dealloco il nodo da rimuovere */

           }

    return OK;

}
```

deleteElemR
(versione ricorsiva)

```
Error tree::deleteElemR(const Label l, Tree& t)

{

/* ....deleteElemR viene sempre chiamata a partire dalla radice;
se t non è vuoto e t->label è uguale a l, vuole dire che voglio
cancellare la radice. Questo è possibile solo se la radice non ha
figli, nel qual caso la rimuovo e restituisco OK, altrimenti
restituisco FAIL: devo gestire questo caso con opportuni
controlli.... */

/* se non sono nella situazione di tentativo di rimozione della
radice, allora chiamo una funzione ausiliaria deleteElemAux
definita ricorsivamente; in deleteElemAux posso fare l'assunzione
di non essere nel caso di tentativo di rimozione della radice, già
trattato prima */

return deleteElemAux(l, t);

}
```

deleteElemAux
(funzione ricorsiva ausiliaria di deleteElemR)

```
Error deleteElemAux(const Label l, Tree& t)

{

  /* ...CASO 1: se t è vuoto, non c'é nulla da cancellare:
restituisco FAIL...*/

   /* ...CASO 2: se t ha un figlio con etichetta l lo rimuovo
(aggiustando tutti i puntatori, tenendo quindi conto del caso se
questo figlio è il primo figlio o è “in mezzo”: si veda anche la
descrizione della versione iterativa) e restituisco OK */

   /* ...CASO 3: se non ho fatto return nel caso 2, vuole dire che
nessuno dei figli di t ha l'etichetta cercata... però i loro
discendenti potrebbero averla! Richiamo quindi deleteElemAux sui
figli di t finché o una delle chiamate ha successo (restituisce OK),
il che vuole dire che ho trovato il nodo da cancellare nel
sottoalbero e lo ho cancellato, nel qual caso posso restituire OK,
oppure non ci sono più figli da esplorare, nel qual caso devo
restituire FAIL */

}
```

Altre operazioni

- **father** si può implementare ricorsivamente come una opportuna variante della visita DFS (cerco il padre di un nodo con una certa etichetta l a partire da t: se t non è il padre del nodo cercato, allora si deve proseguire la ricerca tra i figli di t finché non si trova il risultato).
- **children** è banale: si può usare la getNode, recuperare il primo figlio del nodo trovato e seguire la lista dei suoi fratelli. Analogamente la **degree** è banale in quanto una volta raggiunto un nodo con la getNode, contare i suoi figli è facile.
- **ancestors** si può implementare in modo iterativo e molto semplice “risalendo” la catena dei padri mediante chiamate a father, oppure in modo ricorsivo, più elegante ed efficiente. Nota: le versioni ricorsive spesso sono più inefficienti delle equivalenti versioni iterative, ma in questo caso le numerose chiamate a father rendono la versione iterativa particolarmente inefficiente.
- **leastCommonAncestor** si implementa in modo semplice recuperando gli ancestor di entrambe le etichette e scandendo le liste degli ancestor: la prima etichetta diversa nelle liste indica la “separazione” dei rami; il minimo antenato comune è l'elemento prima di quello diverso
- **member** e **numNodes** sono varianti della ricerca DFS, in cui l'operazione op da effettuare è un confronto oppure un'aggiunta di un'unità a un contatore...