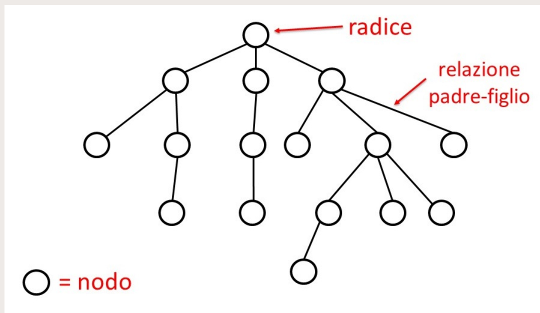
Dati e Algoritmi 1: A. Pietracaprina

Alberi Generali

Alexander Calder, Arc of Petals, 1941. Peggy Guggenheim Collection, Venice.

Nozione (informale) di Albero

Collezione di nodi caratterizzata una struttura gerarchica che si dipana da una nodo radice tramite relazioni di tipo padre-figlio.

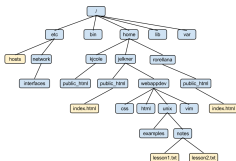


Osservazioni:

- le relazioni padre-figlio costituiscono un insieme di collegamenti minimali che inducono un legame (connessione) tra tutti i nodi;
- Una lista è un caso estremo di albero con una struttura gerarchica lineare.

Campi Applicativi

1. strutture dati: dizionari; code con priorità
2. esplorazione risorse: filesystem; siti di e-commerce;



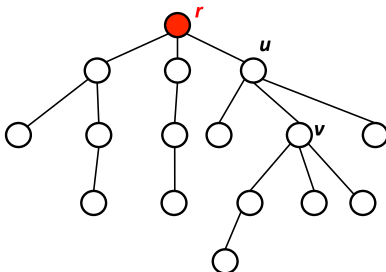
3. sistemi distribuiti e reti di comunicazione: sincronizzazione, broadcast, gathering
4. analisi di algoritmi: albero della ricorsione
5. classificazione: alberi di decisione
6. compressione di dati (codici di Huffman)
7. biologia computazionale: alberi filogenetici

Alberi (Tree) (Capitolo 8 [GTG14])

Definizione di albero radicato (rooted tree)

Un albero radicato T è una collezione di nodi che, se non è vuota, soddisfa le seguenti proprietà:

- $\exists$ un nodo speciale $r \in T$ ($r \doteq$ radice)
- $\forall v \in T, v \neq r : \exists! u \in T$: u è padre di v (v è figlio di u)
- $\forall v \in T, v \neq r :$ risalendo di padre in padre si arriva a r (= ogni nodo è discendente dalla radice.)

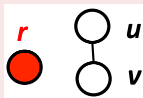


Nota

Nel libro la terza condizione:

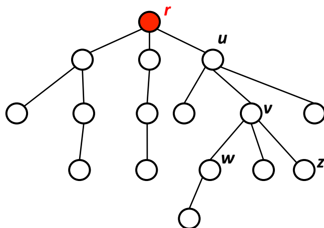
- $\forall v \in T, v \neq r$: risalendo di padre in padre si arriva a r

manca. Senza di questa, la seguente collezione



con u padre di v e v padre di u sarebbe un albero ... che ha poco senso. Questa collezione piuttosto è una foresta di alberi.

Alberi: altre definizioni



Antenati

x è antenato di y se $x = y$ oppure x è antenato del padre di y (es: u è antenato di w

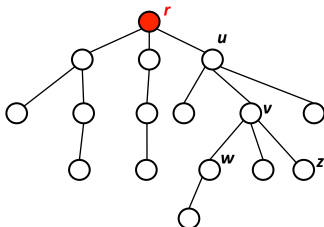
Discendenti

x è discendente di y se y è antenato di x

Nodi interni

Nodi con ≥ 1 figli

Alberi: altre definizioni (continua)



Nodi esterni (= foglie)

Nodi senza figli.

Sottoalbero con radice v

T_v = albero formato da tutti i discendenti di v

Albero ordinato

T è un albero ordinato se per ogni nodo interno $v \in T$ è definito un ordinamento lineare tra i figli $u_1, u_2, \ldots, u_k$ di v.

Definizione Ricorsiva

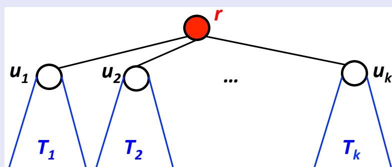
Definizione ricorsiva di albero radicato

Un albero radicato T è una collezione di nodi che, se non è vuota, risulta partizionata in questo modo:

$$T = \{r\} \cup T_1 \cup T_3 \cup \cdots \cup T_k,$$

per un qualche $k \geq 0$, dove:

- r è radice con figli $u_1, u_2, \ldots, u_k$;
- $\forall i, 1 \leq i \leq k$: T_i è un albero non vuoto con radice u_i.



Dove T_i denota il sottoalbero T_{u_i}.

Alberi: ancora definizioni

Profondità di un nodo v in un albero T: $\text{depth}_T(v)$

Due definizioni alternative:

Def.1: $\text{depth}_T(v) = |\text{antenati}(v)| - 1$;

Def.2:

- se $v = r$ radice $\Rightarrow \text{depth}_T(v) = 0$
- altrimenti $\text{depth}_T(v) = 1 + \text{depth}_T(\text{padre}(v))$

Livello i

Insieme dei nodi a profondità i ($\forall i \geq 0$)

Altezza di un nodo v in un albero T: $\text{height}_T(v)$

- se v è foglia $\Rightarrow \text{height}_T(v) = 0$
- altrimenti $\text{height}_T(v) = 1 + \max_{w:w \text{ figlio di } v}(\text{height}_T(w))$

Alberi: ancora definizioni... e una proposizione

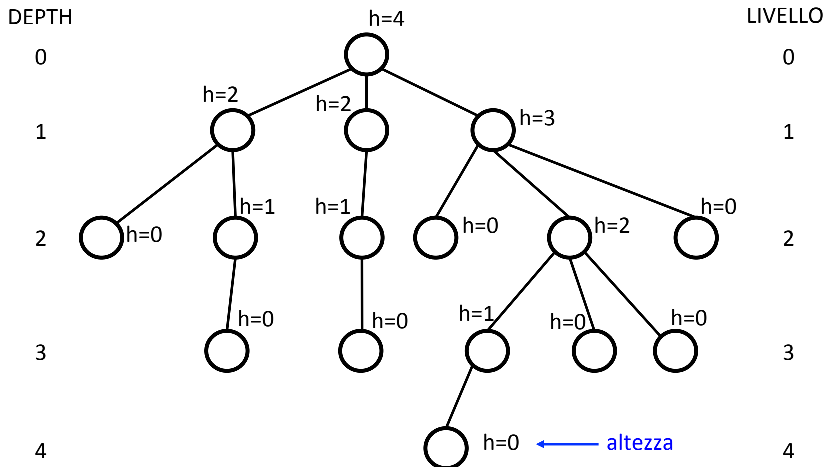
Altezza di un albero T

$\text{height}(T) = \text{height}_T(r)$, con r radice di T

Proposizione (Proposition 8.3 [GTG14])

Dato un albero T, $\text{height}(T) = \max_{v \in T : v \text{ foglia}} (\text{depth}_T(v))$

Esempio



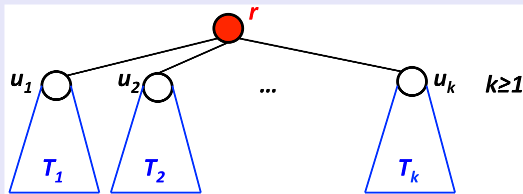
Proposizione (Proposition 8.3 [GTG14])

Dato un albero T, $\text{height}(T) = \max_{v \in T : v \text{ foglia}} (\text{depth}_T(v))$

Dimostrazione (Esercizio R-8.2 [GTG14])

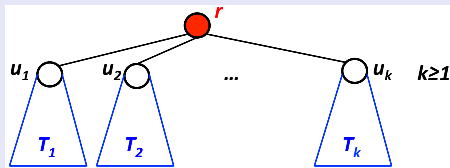
Induzione su n = numero di nodi di T

- base: n=1 $\Rightarrow$ OK
- passo induttivo: fisso $n \geq 1$. Come hp. induttiva assumiamo che la proprietà valga per qualsiasi albero con m nodi, per $1 \leq m \leq n$. Consideriamo un albero T con $n+1$ nodi. Dato che $n+1 > 1$, si ha che T è del tipo:



(Si osservi che $T_i = T_{u_i}$, per $1 \leq i \leq k$.)

Dimostrazione (continua)



$$
\begin{aligned}
\text{height}(T) &\overset{(def)}{=} 1 + \max_{1\le i\le k}(\text{height}_T(u_i)) \\
&= 1 + \max_{1\le i\le k}(\text{height}_{T_i}(u_i)) \\
&\overset{(hp.ind.)}{=} 1 + \max_{1\le i\le k}\left(\max_{\text{foglia } v\in T_i}(\text{depth}_{T_i}(v))\right) \\
&= \max_{1\le i\le k}\left(\max_{\text{foglia } v\in T_i}(1+\text{depth}_{T_i}(v))\right) \\
&= \max_{1\le i\le k}\left(\max_{\text{foglia } v\in T_i}\text{depth}_T(v)\right) \\
&= \max_{\text{foglia } v\in T}\text{depth}_T(v)
\end{aligned}
$$

Interfacce Iterator e Iterable

Prima di definire l'interfaccia `Tree` definiamo due interfacce:

- `Iterator`: un "cursore" che permette di enumerare (scan) gli elementi di una collezione;
- `Iterable`: una collezione che rende disponibile un iteratore ai suoi elementi.

Si veda [GTG14, Paragrafo 7.4] per maggiori dettagli.

```
public interface Iterator<E> {
  /** Returns true if the scan of the collection is not over */
  boolean hasNext();
  /** Returns the next element in the collection */
  E next();
}
public interface Iterable<E> {
  /** Returns an iterator of the collection */
  Iterator<E> iterator()
}
```

Interfaccia Tree

public interface Tree<E> **extends** Iterable<E> {

```
/** Returns the number of positions in the tree */
int size();
/** Returns true if the tree contains no positions */
boolean isEmpty();
/** Returns the Position of the root (or null if empty)*/
Position<E> root();
/** Returns the Position of p's parent (or null if p is the root) */
Position<E> parent(Position<E> p);
/** Returns an iterable containing p's children */
Iterable<Position<E>> children(Position<E> p);
/** Returns the number of children of p */
int numChildren(Position<E> p);
....
```

```
  ....
  /** Returns true if p is internal */
  boolean isInternal(Position<E> p);
  /** Returns true if p is external */
  boolean isExternal(Position<E> p);
  /** Returns true if p is root */
  boolean isRoot(Position<E> p);
  /** Returns an iterator to all element in the tree */
  Iterator<E> iterator();
  /** Returns an iterable containing all positions in the tree */
  Iterable<Position<E>> positions();
}
```

Osservazioni:

- `iterator()` deriva dal fatto che `Tree<E>` estende `Iterable<E>`
- Assumiamo complessità $\Theta(1)$ per tutti i metodi, tranne `children` `iterator` e `positions`, e che sia possibile enumerare i figli di un nodo (tramite `children`) in tempo proporzionale al loro numero

Calcolo della profondità di un nodo (Algoritmo ricorsivo)

Algoritmo: `depth(`T`,`v`)`

Input: $v \in T$
Output: profondità di v in T
if (T`.isRoot(`v`)`) **then return** 0;
else return $1+$`depth(`T`,`T`.parent(`v`))`;

Osservazione

[GTG14] definisce gli algoritmi di base per gli alberi come metodi di una classe astratta che implementa l'interfaccia `Tree`, e non specifica l'albero T come parametro in quanto esso è implicitamente associato all'istanza da cui si invoca il metodo (`this`).

Complessità di depth

Consideriamo l'albero della ricorsione associato all'esecuzione di depth(T,v), per un nodo v arbitrario di profondità d_v. È facile vedere che

- L'albero della ricorsione ha $d_v + 1$ nodi corrispondenti alle invocazioni ricorsive dell'algoritmo sui $d_v + 1$ antenati di v (incluso v)
- Il costo associato a ciascun nodo dell'albero della ricorsione è $\Theta(1)$.

Di conseguenza, la complessità di depth(T,v) è $\Theta(d_v + 1)$.

Osservazione: Se volessimo esprimere la complessità in funzione del numero di nodi n di T essa sarebbe $\Theta(n)$ dato che esistono alberi di n nodi con nodi a profondità $\Theta(n)$. Tuttavia, usare la profondità del nodo come taglia dell'istanza caratterizza la complessità in modo più accurato.

Calcolo della profondità di un nodo (Algoritmo iterativo)

Algoritmo: `iter-depth(`T`,`v`)`

Input: $v \in T$
Output: profondità di v in T

```
d ← 0 ;
while !(T.isRoot(v)) do
    v ← T.parent(v);
    d ← d + 1
return d
```

Osservazione: Ha la stessa complessità dell'algoritmo ricorsivo ($\Theta(d_v + 1)$)

Calcolo dell'altezza di un nodo

Algoritmo: `height(`T`,`v`)`

Input: $v \in T$
Output: altezza di v in T
$h \leftarrow 0$;
foreach $w \in T$`.children(`v`)` **do** $h \leftarrow \max\{h, 1+$`height(`T`,`w`)`$\}$;
return h

Complessità di height

Consideriamo l'albero della ricorsione associato all'esecuzione di `height`(T, v), per un nodo v arbitrario. Si osserva che

- L'algoritmo viene invocato esattamente una volta su ogni nodo $u \in T_v$. Di conseguenza, l'albero della ricorsione ha T_v nodi corrispondenti alle invocazioni ricorsive dell'algoritmo.
- Il costo associato al nodo dell'albero della ricorsione corrispondente alla invocazione `height`(T, u) per un qualche $u \in T_v$ è $\Theta(c_u + 1)$, dove c_u è il numero di figli di u. (Si contano le operazioni eseguite da `height`(T, u), tranne quelle delle invocazioni ricorsive al suo interno.)
- Se T_v contiene n nodi ($n = |T_v|$), si ha che

$$\sum_{u \in T_v} c_u = n - 1 \qquad \text{(Proposizione 8.4 [GTG14])}$$

Si dimostra osservando che ogni nodo di T_v tranne la radice v ha un padre (unico) in T_v e contribuisce 1 alla sommatoria.

Complessità di height (continua)

Dai punti precedenti deduciamo che la complessità di `height`(T, v) è

$$\Theta\left(\sum_{u \in T_v} (c_u + 1)\right) = \Theta(2n - 1) = \Theta(n),$$

dove $n = |T_v|$.

Osservazione: Si noti che per entrambi gli algorithmi `depth` e `height` vale che tutte le istanze della stessa taglia (profondità d del nodo v per `depth` e numero di nodi n in T_v per `height`) richiedono lo stesso numero di operazioni.

Visite di Alberi

Visita

Scansione sistematica tutti i nodi dell'albero che permette di eseguire una qualche operazione (visita) ad ogni nodo.

Vedremo le seguenti visite:

- Preorder: visita prima il padre e poi (ricorsivamente) i sottoalberi radicati nei figli.
- Postorder: visita prima (ricorsivamente) i sottoalberi radicati nei figli, poi il padre.

Visita in Preorder

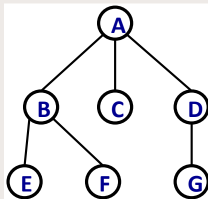
Algoritmo `preorder(`T`,`v`)`
Input: nodo $v \in T$
Output: visita di tutti i nodi di T_v
visita v;
foreach $w \in T$`.children(`v`)` **do**
  `preorder(`T`,`w`)`

Chiamata iniziale

`preorder(`T`,`T`.root())`

Nota

Se l'albero è ordinato la visita tocca i figli di un nodo nell'ordine dato.



A B E F C D G

Visita in Postorder

Algoritmo `postorder(`T`,`v`)`
Input: nodo $v \in T$
Output: visita di tutti i nodi di T_v
foreach $w \in T$`.children(`v`)` **do**
  `postorder(`T`,`w`)`

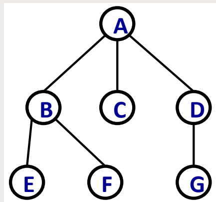
visita v;

Chiamata iniziale

`postorder(`T`,`T`.root())`

Nota

Se l'albero è ordinato la visita tocca i figli di un nodo nell'ordine dato.



E F B C G D A

Complessità delle visite

Determiniamo la complessità di `preorder(T,T.root())` e `postorder(T,T.root())`, in funzione del numero di nodi n di T.

L'analisi è simile a quella dell'algoritmo `height`:

- L'albero della ricorsione ha n nodi, che corrispondono alle invocazioni ricorsive dell'algoritmo su ciascun $u \in T$.
- Il costo associato al nodo dell'albero della ricorsione corrispondente alla invocazione su $u \in T$ è

$$\Theta\left(c_u + 1 + t_u\right),$$

  dove c_u è il numero di figli di u e t_u è il costo della "visita" di u.
- La complessità totale è

$$\Theta\left(\sum_{u \in T}(c_u + 1 + t_u)\right) = \Theta\left(n + \sum_{u \in T} t_u\right).$$

  Si osservi che se $t_u \in O(1)$ per ogni u, o più in generale $\sum_{u \in T} t_u \in O(n)$, la complessità delle visite è $\Theta(n)$.

Definizione

Sia T albero ordinato e siano $u, v \in T$ due nodi allo stesso livello. Diciamo che

u è *a sinistra* di v ($\Rightarrow$ v è *a destra* di u)

se u viene prima di v nella visita in preorder.

NB: si assume che nella visita in preorder i figli di un nodo siano visitati secondo l'ordine a loro assegnato

Osservazione La definizione è coerente con il modo di disegnare gli alberi.

Visite come design pattern

Le visite degli alberi rappresentano dei *design pattern algoritmici* che possono essere istanziate per risolvere diversi problemi che mirano al calcolo di determinati valori e/o all'impostazione di opportune variabile associate ai nodi.

In particolare:

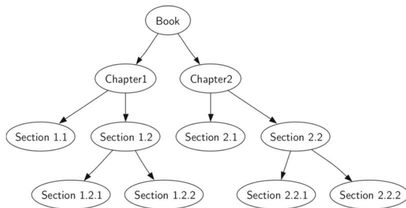
- **Visita in preorder**: il contributo di un nodo nell'algoritmo è funzione di quello dei suoi antenati.
- **Visita in postorder**: il contributo di un nodo nell'algoritmo è funzione di quello dei suoi discendenti.

Esempi di Visite

Esempi di visita in preoder

Esempio 1: Si vuole stampare l'indice di un libro la cui struttura è rappresentata da un albero, dove

- i nodi rappresentano le sezioni del libro (capitoli, paragrafi, ecc.)
- la "visita" di un nodo consiste nella stampa dell'identificatore della sezione corrispondente.



La **visita in preorder** dell'albero in figura stampa la sequenza:

Book: Chapter1, Section 1.1., Section 1.2, Section 1.2.1, Section 1.2.2, Chapter 2, Section 2.1, Section 2.2, Section 2.2.1, Section 2.2.2

Esempio analogo: stampa della struttura di un file system come sequenza di cartelle e file.

Esempi di visita in preoder

Esempio 2: vogliamo progettare un algoritmo che, dato un albero T faccia le seguenti cose: per ogni nodo $v \in T$ calcoli la sua profondità e la memorizzi in un campo v`.depth`.

Si può adattare la visita in preorder, definendo la visita di un nodo v in questo modo: se v è la radice, la sua profondità viene impostata a 0, altrimenti si imposta la profondità a 1+la profondità del padre, che è già impostata, dato che il padre è stato già visitato.

Algoritmo `AllDepths(`T`,`v`)`
Input: nodo $v \in T$ (il padre, se esiste, ha depth impostato)
Output: tutti i nodi di T_v con campo depth impostato
if $(T.\mathrm{isRoot}(v))$ **then** v.depth $\leftarrow$ 0;
else v.depth $\leftarrow 1+T$.parent(v).depth;
foreach $w \in T$`.children(`v`)` **do** `AllDepths(`T`,`w`)`;

Prima invocazione: `AllDepths(`T`,`T`.root())`

Complessità: $\Theta(n)$, dato che l'algoritmo ha la stessa struttura della visita in preorder e la "visita" di un nodo (impostazione del campo `depth`) richiede $O(1)$ operazioni.

Esempi di visita in postorder

Esempio 1: l'algoritmo `Height` è un esempio di visita in postorder dato che l'altezza di un nodo viene calcolata solo dopo aver calcolato quelle dei figli.

Esempio 2: Si consideri un file system gerarchico la cui struttura è rappresentata da un albero T dove i nodi interni corrispondono alle cartelle e i nodi foglia ai file. Ogni nodo v ha un campo v`.loc-size` che memorizza lo spazio occupato dal nodo, escludendo quello dei discendenti.

Vogliamo progettare un algoritmo ricorsivo (`DiskSpace`) che per ogni nodo $v \in T$ calcoli lo spazio aggregato occupato dai suoi discendenti e lo memorizzi in un campo v`.aggr-size`.

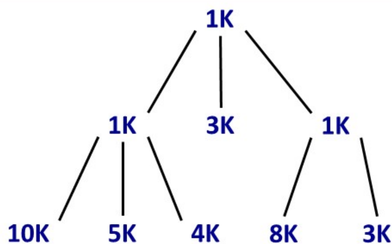
Algoritmo `DiskSpace(`T`,`v`)`
Input: nodo $v \in T$
Output: spazio aggregato dei nodi di T_v aggiornandone i campi aggr-size

```
v.aggr-size ← v.loc-size ;
foreach w ∈ T.children(v) do
    v.aggr-size ← v.aggr-size+DiskSpace(T,w)
return v.aggr-size;
```

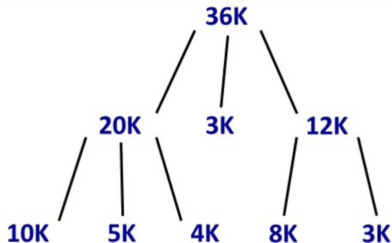
Prima invocazione: `DiskSpace(`T`,`T`.root())`

Complessità: $\Theta(n)$ (dove n è il numero di nodi dell'albero), dato che l'algoritmo ha la stessa struttura della visita in postorder e la "visita" di un nodo (impostazione del campo `aggr-size`) richiede $O(1)$ operazioni.

Albero T con i valori dei campi loc-size



Albero T con i valori dei campi aggr-size dopo l'esecuzione di DiskSpace(T,T.root())



Esercizi

Esercizio

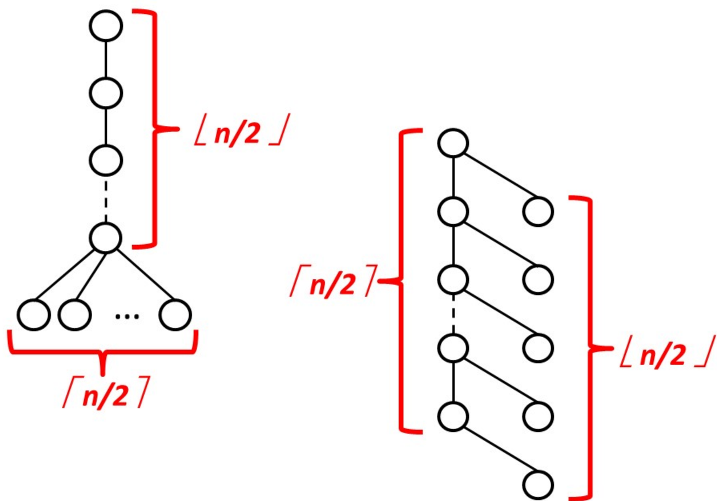
Si supponga di calcolare l'altezza di un albero T di n nodi invocando l'algoritmo $\texttt{depth}(T, v)$ da ciascuna foglia v di T e restituendo come altezza la massima profondità ottenuta. Dimostrare che tale strategia ha una complessità al caso pessimo $\Omega\left(n^2\right)$. (È l'algoritmo `heightBad` decritto in [GTG14]. Si veda anche l'esercizio C-8.27 del testo.)

Svolgimento

Dato che il calcolo della profondità d_v di un nodo v richiede $\Theta\left(d_v + 1\right)$ operazioni, la complessità della strategia indicata è

$$\Omega\left(\sum_{v:v \text{ foglia}} (1 + d_v)\right)$$

Gli esempi nel prossimo lucido mostrano che per ogni n esiste un albero di n nodi in cui $\sum_{v:v \text{ foglia}}(1 + d_v) \in \Omega\left(n^2\right)$.



Nell'albero a sx ci sono $\lceil n/2 \rceil$ foglie a profondità $\Theta(n)$, mentre in quello a dx le $\lceil n/4 \rceil$ foglie più profonde sono sicuramente a profondità $\Theta(n)$.

Esercizio 4 (C-8.50 [GTG14])

Sia T un albero. Dati due nodi $v, w \in T$ si definisce il *Lowest Common Ancestor* di v e w (LCA(v, w)) come l'antenato comune più profondo. Progettare un algoritmo efficiente per trovare LCA(v, w), analizzandone la complessità.

Si osservi che un antenato comune esiste sempre, ed è la radice.

Svolgimento

Idea:

- Determinare le profondità di v e w usando l'algoritmo `depth` visto in precedenza.
- Risalire dal più profondo tra v e w sino all'antenato che sta alla stessa profondità dell'altro, e da qui risalire da entrambi sino a trovare il primo antenato comune.

Pseudocodice:

```
Algoritmo LCA(T, v, w)
input v, w ∈ T
output Least Common Ancestor di v e w

d_v ← depth(T, v); d_w ← depth(T, w)
if (d_v > d_w) then
   for i ← 1 to d_v − d_w do v ← T.parent(v)
else
   for i ← 1 to d_w − d_v do w ← T.parent(w)
while (v ≠ w) do {
   v ← T.parent(v)
   w ← T.parent(w)
}
return v
```

Complessità:

- Le invocazioni di `depth` hanno complessità proporzionale alle profondità di v e w.
- I cicli **for** e il ciclo **while** eseguono, ciascuno, un numero di iterazioni limitato superiormente dalla massima profondità dei due nodi, e in ciascuna iterazione eseguono un numero costante di operazioni.

Dato che la profondità di un nodo è limitata superiormente dall'altezza dell'albero, concludiamo che la complessità dell'algoritmo è $O(h)$, con h altezza di T.

Esercizio: Dimostrare che la complessità è $\Theta(h)$.

Osservazione

Le complessità di algoritmi su alberi si esprimono spesso in funzione del numero di nodi n o dell'altezza h. Si noti che una complessità espressa in funzione di h può essere anche espressa in funzione di n usando il fatto che $0 \leq h < n$.

Esercizio (simile a R-8.19 di [GTG14])

Si supponga che la visita in preorder di un albero ordinato di 6 nodi incontri i nodi nell'ordine ABCDEF.

1. Dire quali delle seguenti sequenze può rappresentare la visita in postorder dello stesso albero (motivando la risposta): BAFECD, CDBFEA, CDAEFB.
2. Disegnare l'albero compatibile con le due sequenze di preorder e postorder.

Esercizio

Progettare un algoritmo che dato un albero T, per ciascun nodo $v \in T$ memorizzi la sua altezza in un campo v`.height`, e analizzarne la complessità.

Errata

Cambiamenti rispetto alla prima versione dei lucidi:

- Lucido 9: chiarita la notazione T_i.
- Lucido 15: nuovo lucido su Iterator e Iterable.
- Lucidi 16 e 17 (prima 15 e 16): fatte piccole modifiche al wording.
- Lucidi 19 e 20 (prima 18 e 19): sostituito d con d_v per denotare la profondità di v.