Dati e Algoritmi: A. Pietracaprina

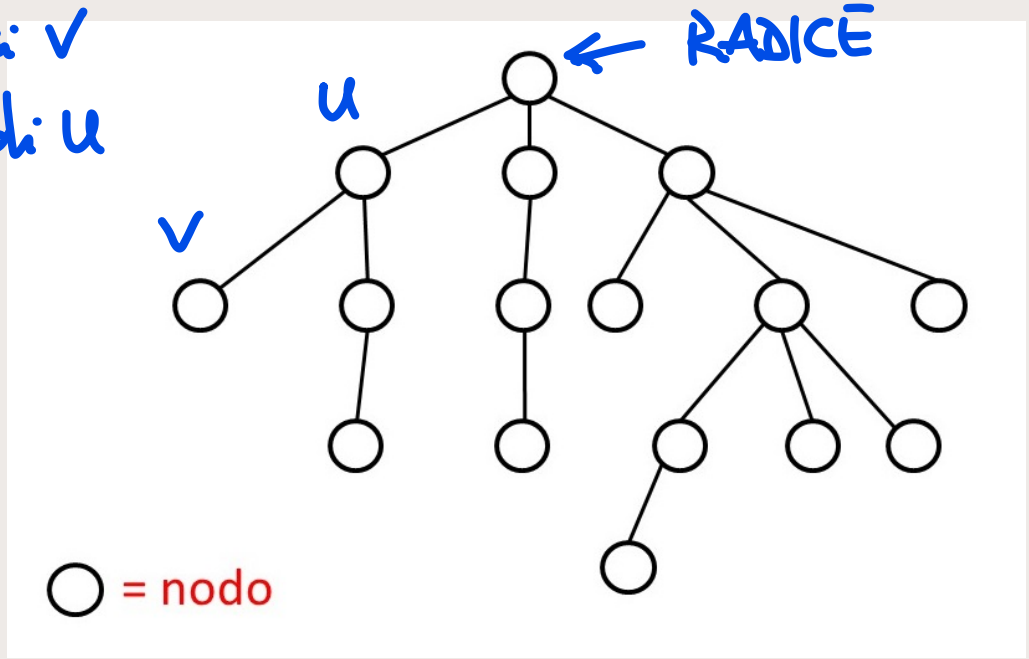
Alberi Generali

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.

u padre di v
v figlio di u

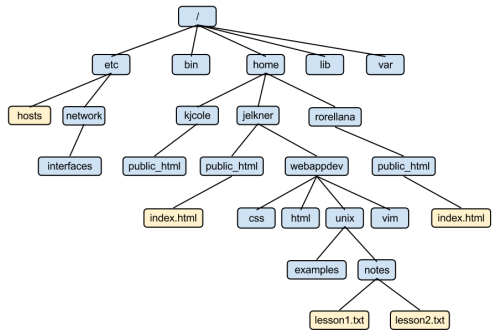


Osservazioni:

- 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: mappe; priority queue
2. esplorazione risorse: filesystem; siti di e-commerce;



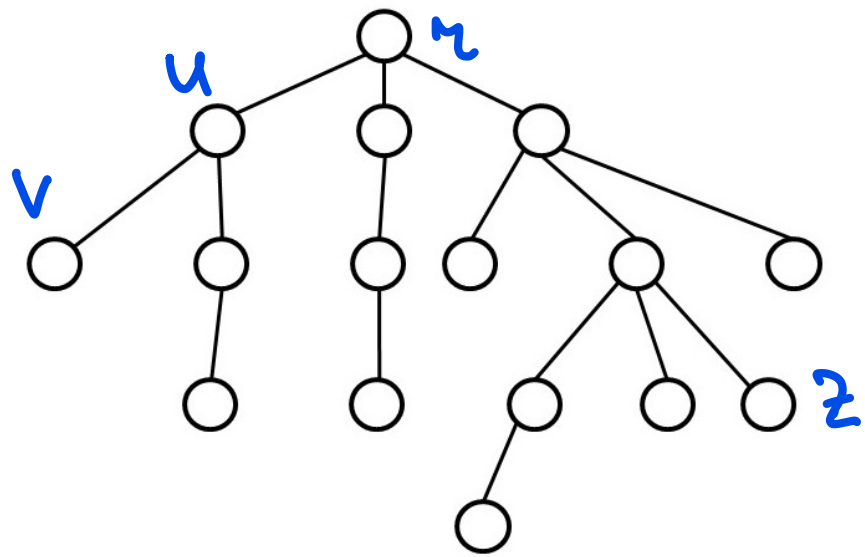
3. sistemi distribuiti e reti di comunicazione: sincronizzazione, broadcast, gathering
4. analisi di algoritmi: albero della ricorsione
5. classificazione: alberi di decisione
6. compressione di dati (codici di Huffman)
7. biologia computazionale: alberi filogenetici

Alberi (Tree)

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.)

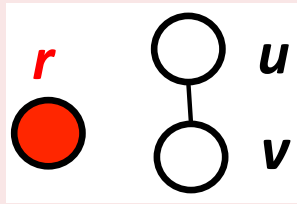


Nota

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

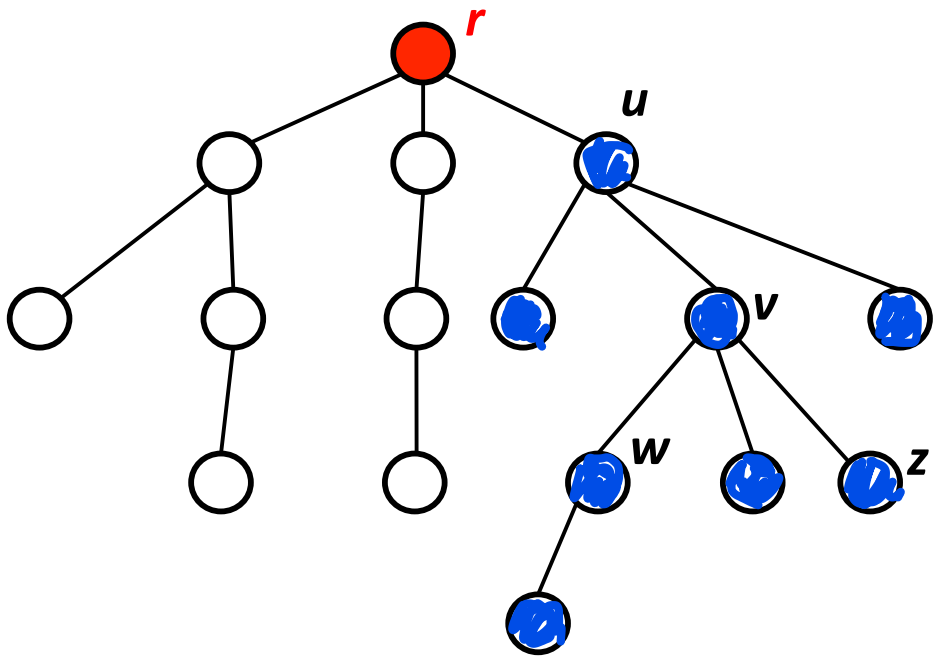
x è antenato di y se $x = y$ oppure x è antenato del padre di y

Discendenti

x è discendente di y se y è antenato di x

Nodi interni

Nodi con ≥ 1 figli



Antenati di w : w, v, u, r

Discendenti di u : nodi

Alberi: altre definizioni

Nodi esterni (= foglie)

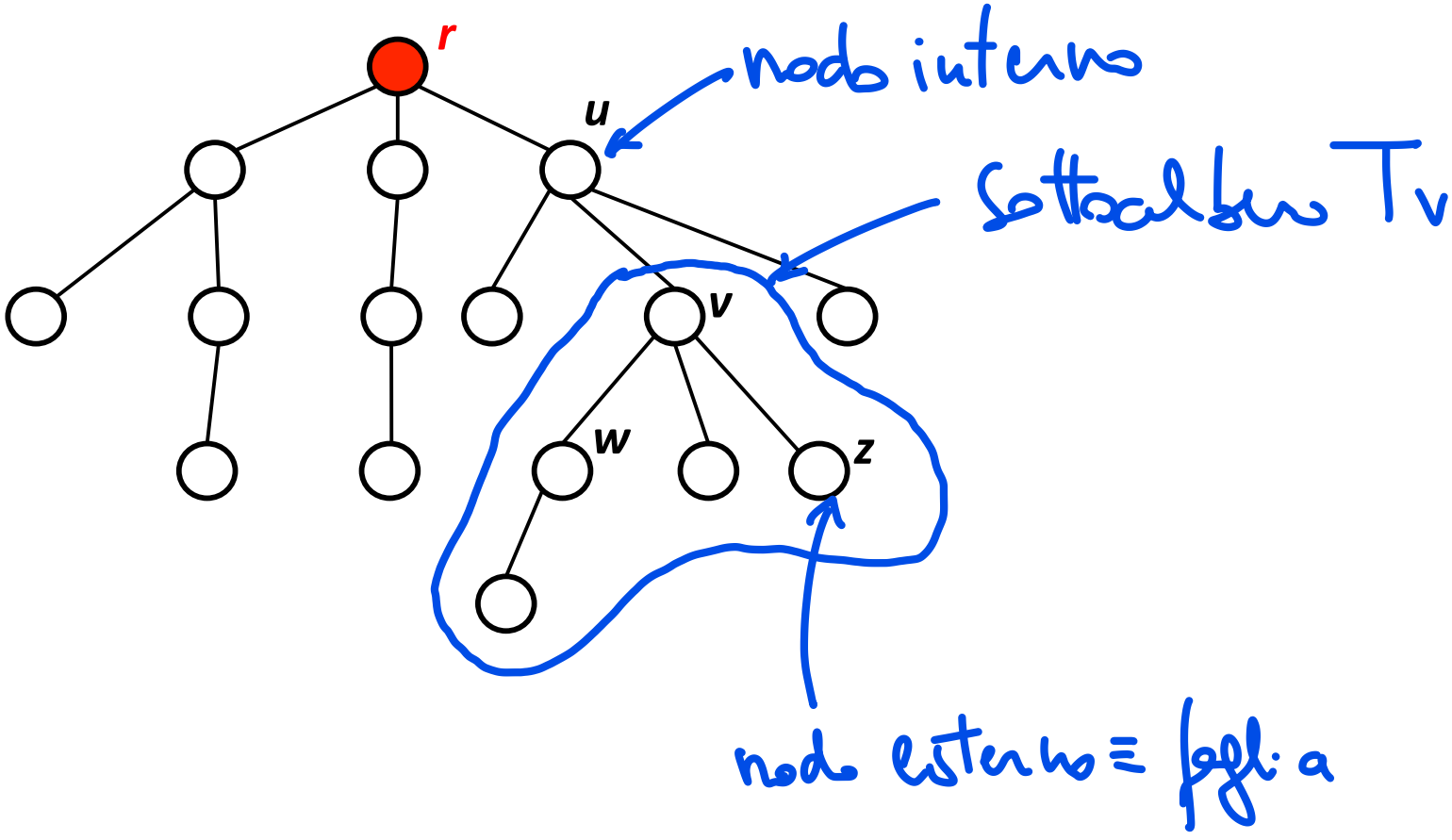
Nodi senza figli.

Sottoalbero con radice v

T_v = albero formato da tutti i discendenti di v (quindi include 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.


r
u
v
w
z
nodo interno
sottoalbero T_v
nodo esterno ≡ foglia

Definizione Ricorsiva

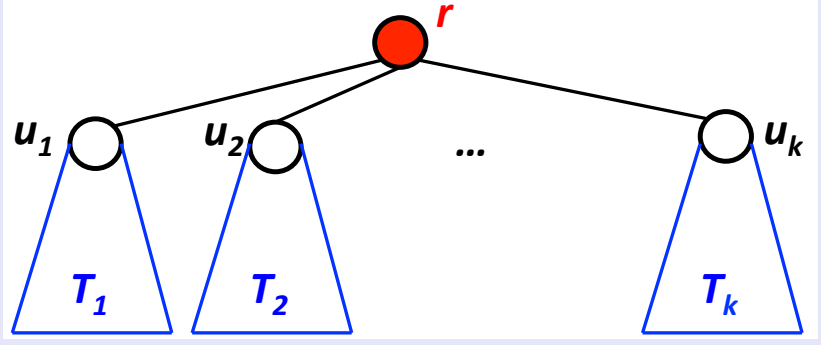
Definizione ricorsiva di albero radicato

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

$$T = \{r\} \cup T_1 \cup T_2 \cup \cdots \cup T_k,$$

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 ($\Rightarrow T_i \equiv T_{u_i}$).



Oss.: le 2 definizioni di albero radicato (ricorsiva e non) sono equivalenti.

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

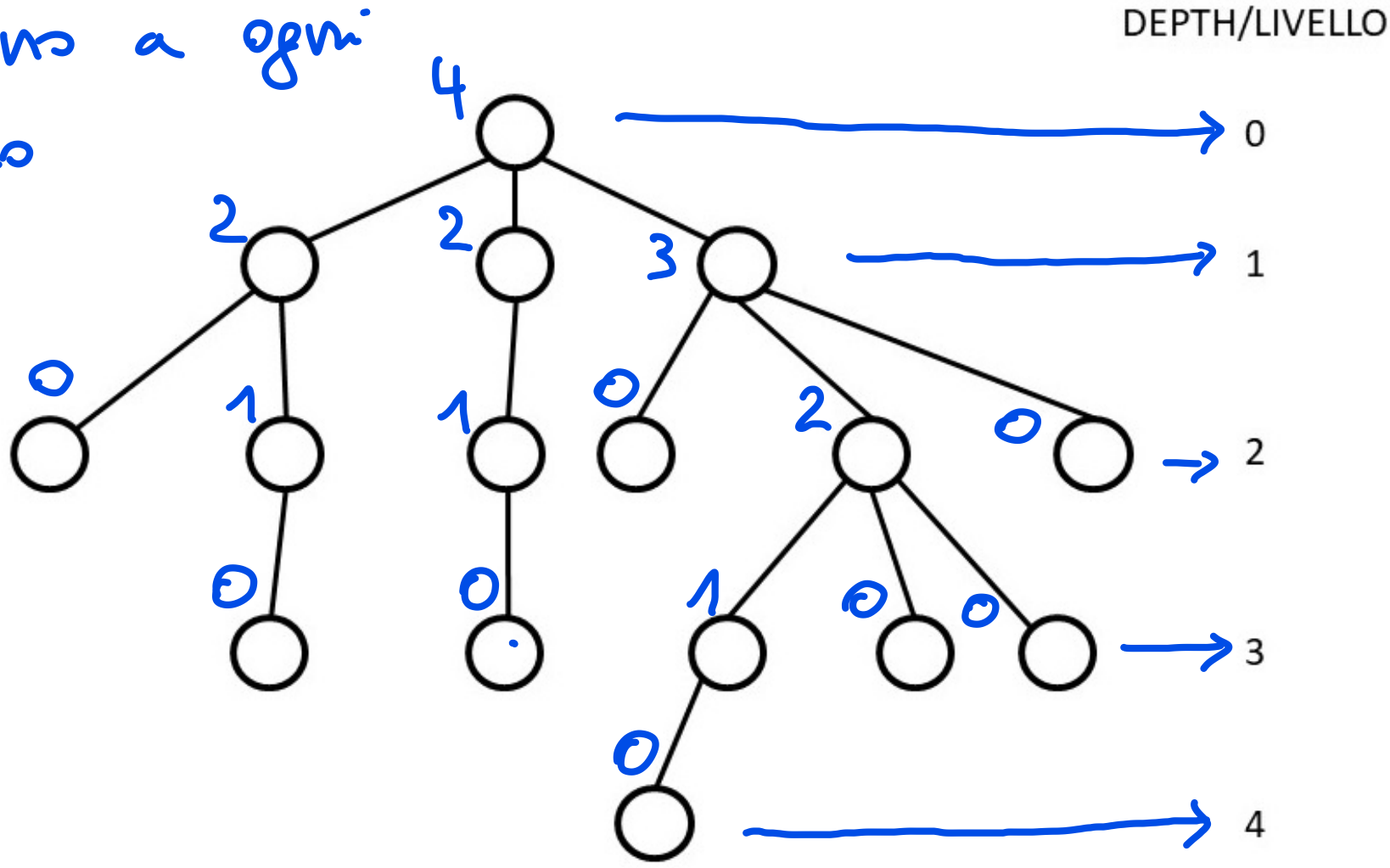
$\text{height}(T) = \text{height}_T(r)$, con r radice di T

Proposizione (Proposition 8.3 [GTG14])

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

Esempio

Le altezze sono indicate vicino a ogni nodo



Proposizione (Proposition 8.3 [GTG14])

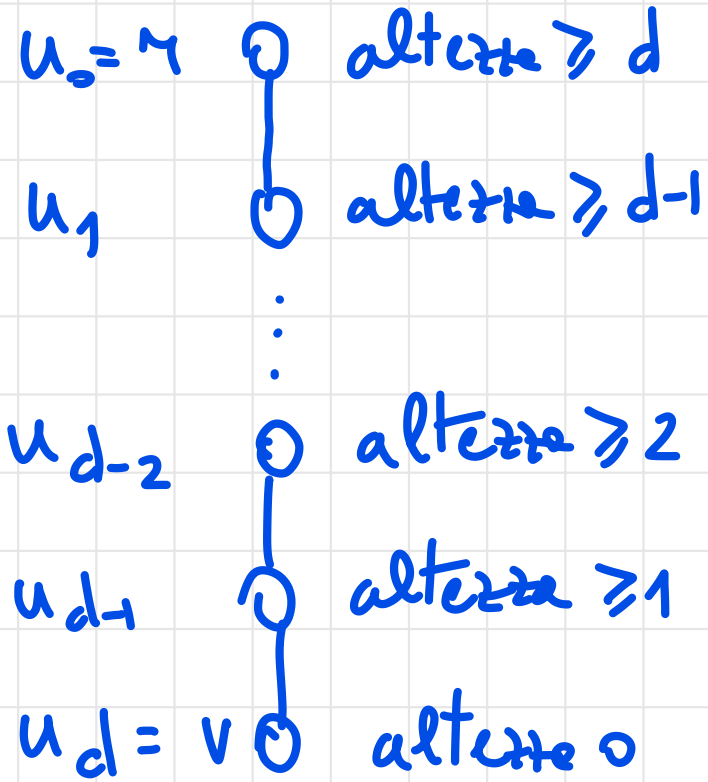
Dato un albero T non vuoto, $\text{height}(T) = \max_{v \in T : v \text{ foglia}} (\text{depth}_T(v))$

Dimostrazione. Definiamo le seguenti quantità

* $h = \text{height}(T)$
* $d = \max_{v \in T : v \text{ foglia}} (\text{depth}_T(v))$

Proviamo: (1) $h \geq d$ e (2) $h \leq d$

(1) Sia v una foglia a profondità d, e sia r la radice di T. Allora esiste un percorso da d a r in T



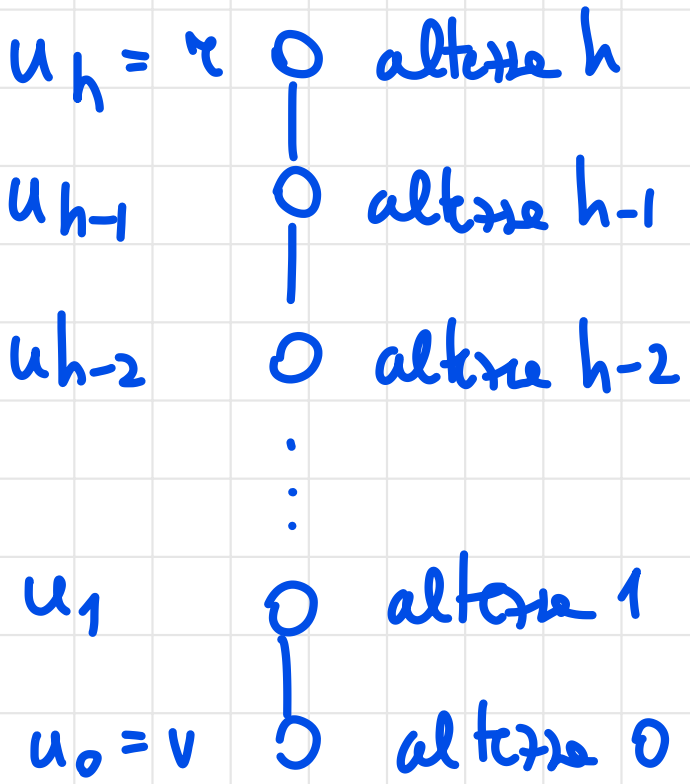
È immediato vedere che

$height_T(u_i) \geq d-i \quad \forall d \geq i \geq 0$

e quindi si ha che

$h = height_T(r) = height_T(u_0) \geq d$

(2) Dimostriamo adesso che $h \leq d$. Per assurdo, se $h > d$ deve esistere un percorso in T dalla radice r a una foglia v



In questo caso, la foglia v ha h antenati $\neq v$, che sono $u_1, u_2, \dots, u_h$, e quindi $\text{depth}_T(v) = h > d$, che contraddice il fatto che d è la massima profondità di una foglia

□

Esercizio : Fare una dimostrazione alternativa della proposizione, per induzione sul numero n di nodi di T $(n \geq 1)$

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

Implementazione tipica di una albero

* Puntatore alla radice (≡ UNICO PUNTO di ACCESSO)
* Ogni nodo v è implementato come un oggetto a sè stante (ad es. di una classe che implementa l'interfaccia Position) che offre dei metodi per accedere al padre e ai figli

  parent(v) ≡ padre di v

  children(v) ≡ figli di v

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, determinando il "prossimo" figlio in tempo costante

Calcolo della profondità di un nodo (Algoritmo ricorsivo)

Algoritmo: `depth(`v`)` // L'albero T è implicito

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

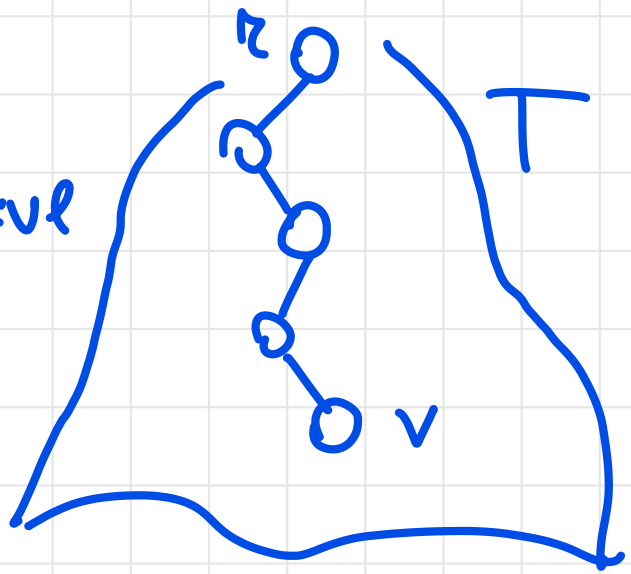
Osservazione

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

COMPLESSITÀ di depth

Albero delle ricorsioni allocato a depth(v) $v \in T$

Se d_v = profondità di v



* $d_v + 1$ invocazioni ricorsive di depth, una per ogni antenato di v
* $\Theta(1)$ operazioni per ogni invocazione ricorsiva

$\Rightarrow$ L'albero delle ricorsioni è costituito da $d_v + 1$ nodi di costo $\Theta(1)$ ciascuno

$\Rightarrow$ Complessità di depth(v) $\in \Theta(d_v + 1)$

Osservazione 1. Il "+1" è giustificato dal fatto che se $d_v = 0$ ($\Rightarrow$ v è radice) comunque bisogna richiedere $\Theta(1)$ operazioni.

TUTTAVIA per semplificare la notazione, scriveremo $\Theta(d_v)$ intendendo $\Theta(d_v + 1)$

Osservazione 2. Se volessimo esprimere la complessità in funzione del numero n di nodi di T, che forma avrebbe? Notazione: $|T|$ = # nodi di T

Usando la profondità d come taglia dell'istanza: le possibili istanze di taglia d sono i nodi a profondità d di tutti i possibili alberi T (di qualsiasi cardinalità |T|)

⇒ Complessità al caso peggiore: $\Theta(d)$

Usando il numero di nodi n come taglia dell'istanza: le possibili istanze di taglia n sono tutti i nodi appartenenti ad alberi T con $|T| = n$

⇒ Complessità al caso peggiore: $\Theta(n)$

Dimostriamo che la complessità è $\Theta(n)$

UPPER BOUND: $O(h)$ perché in un albero con n nodi ogni nodo v ha profondità $\leq n \Rightarrow$ depth(v) richiede $O(n)$ operazioni.

LOWER BOUND: $\Omega(n)$ perché in un albero che è una catena di n nodi, l'ultimo nodo della catena è una foglia v a profondità $n-1$, e quindi depth(v) richiede $\Omega(n)$ operazioni

VERSIONE ITERATIVA di depth

Algoritmo depthITER(v)

input/output: uguali a quelli di depth

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

Complessità $\in \Theta(dv)$

Analisi: esercizio

Calcolo dell'altezza di un nodo

Algoritmo: `height(`v`)`

CASO BASE: v foglia

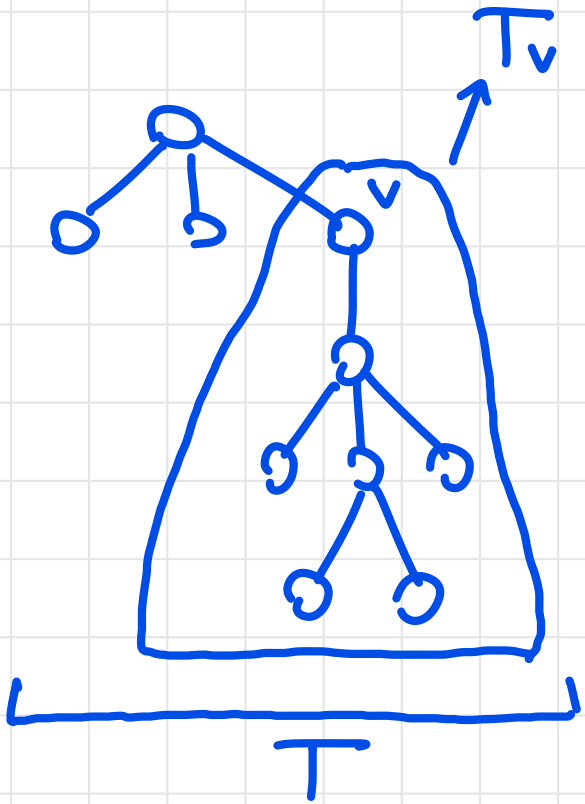
Input: $v \in T$
Output: altezza di v in T
$h \leftarrow 0$;
foreach $w \in T$`.children(`v`)` **do** $h \leftarrow \max\{h, 1+$`height(`w`)`$\}$;
return h

Tecnicamente, T.children(v) è una collezione "Iterable" che contiene i figli di v. Con "foreach w∈T.children(v)" si enumerano i figli di v, ciascuno dei quali viene generato in tempo $\Theta(1)$

COMPLESSITÀ di height(v)

L'albero della ricorsione associato a height(v) con $v \in T$ ha

* Un nodo (cioè una invocazione ricorsiva) per ogni $u \in T_v$
* Costo associato al nodo $u \in T_v$ è $\Theta(c_u + 1)$ dove c_u = # figli di u



$\Rightarrow$ Complessità di height(v) $\in \Theta\left(\sum_{u \in T_v} (c_u + 1)\right)$

Sia n_v il numero di nod. in T_v ($\Rightarrow$ # discendenti di v)

Riscriviamo $\sum_{u \in T_v} (c_u + 1)$ in funzione di n_v

$$\sum_{u \in T_v} (c_u + 1) = \left(\sum_{u \in T_v} c_u \right) + \sum_{u \in T_v} 1$$

$$= \left(\sum_{u \in T_v} c_u \right) + n_v$$

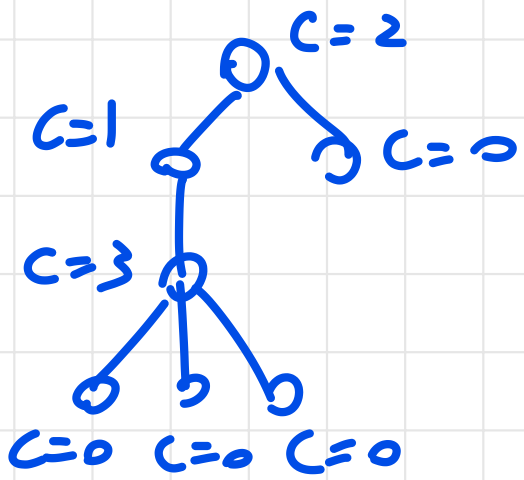
la Proposizione 8.4 del testo dice che $\sum_{u \in T_v} c_u = n_v - 1$

$$\Rightarrow \sum_{u \in T_v} (c_u + 1) = 2n_v - 1$$

$\Rightarrow$ Complessità di height$(v) \in \Theta(n_v)$

Dimostrazione che $\sum_{u \in T_v} c_u = n_v - 1$

Esempio: $n_v = 7$



$$\Rightarrow \sum_{u \in T_v} c_u = 2+1+3 = 6 = n_v - 1$$

In generale la relazione è vera perché ogni nodo di T_v, tranne v, è calcolato ESATTAMENTE una volta in $\sum_{u \in T_v} c_u$ come figlio di suo padre

Dall'analisi fatta sopra discende che la complessità per calcolare l'altezza di tutto l'albero T (invocando height(r) con r radice di T) è $\Theta(n)$, con $n = |T|$, quindi lineare nel numero di nodi dell'albero

Visite di Alberi

Visita di un albero T

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

Rappresentano un *design pattern algoritmico* che può essere istanziato per il calcolo di valori e/o per impostazione opportune variabili associate ai nodi.

Per alberi generali studiamo le seguenti visite:

- Preorder: *visita prima il padre e poi (ricorsivamente) i sottoalberi radicati nei figli*. In questo modo le operazioni svolte per un nodo possono dipendere da quelle svolte per i suoi antenati.
- Postorder: *visita prima (ricorsivamente) i sottoalberi radicati nei figli, poi il padre*. In questo modo le operazioni svolte per un nodo possono dipendere da quelle svolte per i suoi discendenti.

Visita in Preorder: pattern algoritmico

Algoritmo `preorder(`v`)`
Input: nodo $v \in T$
Output: risultante dalla visita di T_v

visita v;

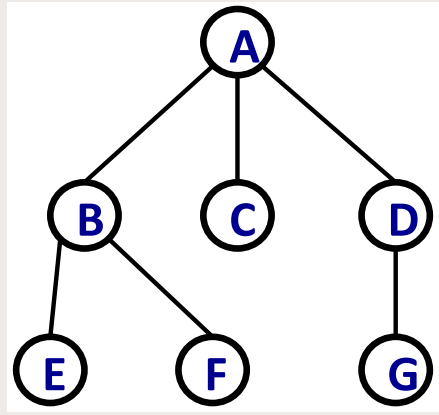
foreach $w \in T$`.children(`v`)` **do**
  `preorder(`w`)`

Chiamata iniziale per vistare T

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

CASO BASE : v è foglia

Esempio



Assumendo che `children()` esamini i figli da sx $\rightarrow$ dx, l'ordine della visita in preorder è

A B E F C D G

Visita in Postorder: pattern algoritmico

Algoritmo `postorder(`v`)`
Input: nodo $v \in T$
Output: risultante dalla visita di T_v

foreach $w \in T$`.children(`v`)` **do**
  `postorder(`w`)`

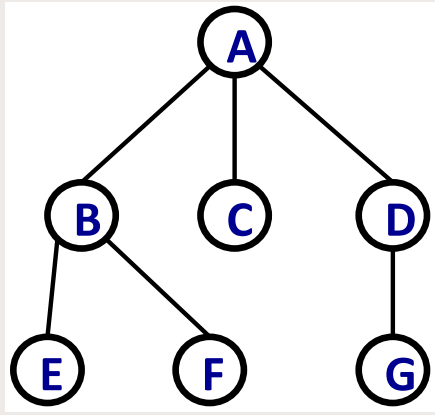
visita v;

Chiamata iniziale per vistare T

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

CAB BASE: v foglia

Esempio



Assumendo che `children()` esamini i figli da sx $\rightarrow$ dx, l'ordine della visita in preorder è

EFBCGDA

Definizione

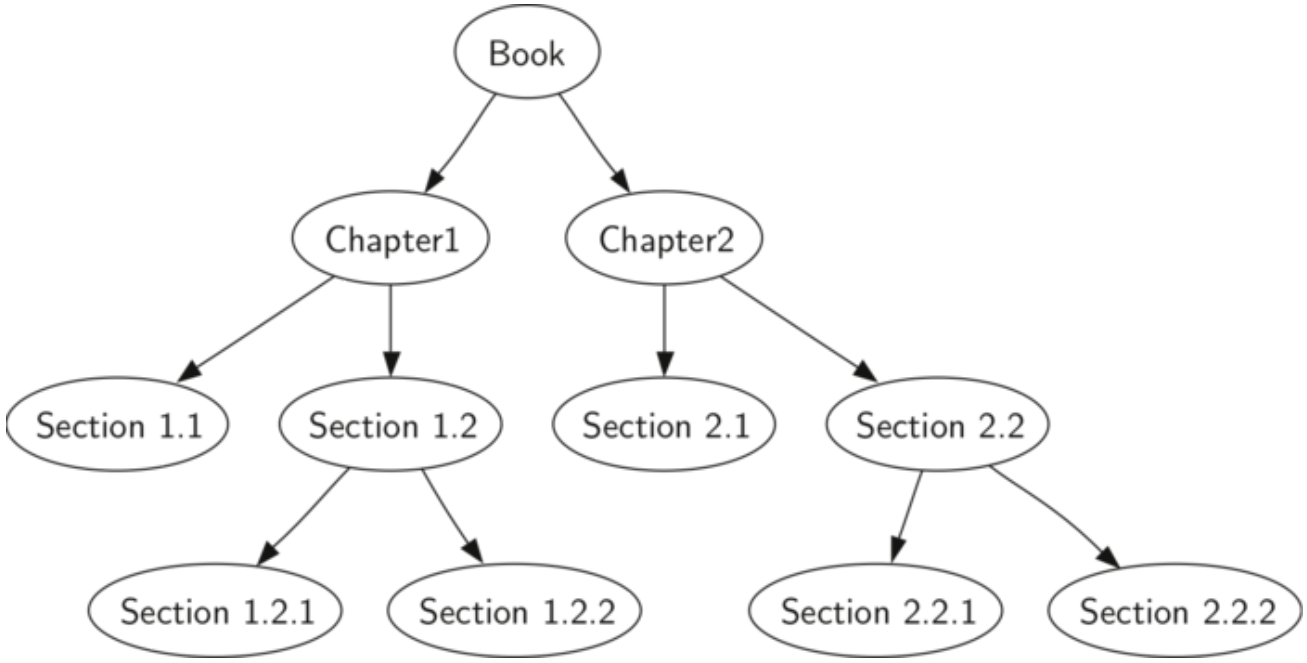
Sia T albero ordinato e siano $u, v \in T$ due nodi allo stesso livello. Diciamo che

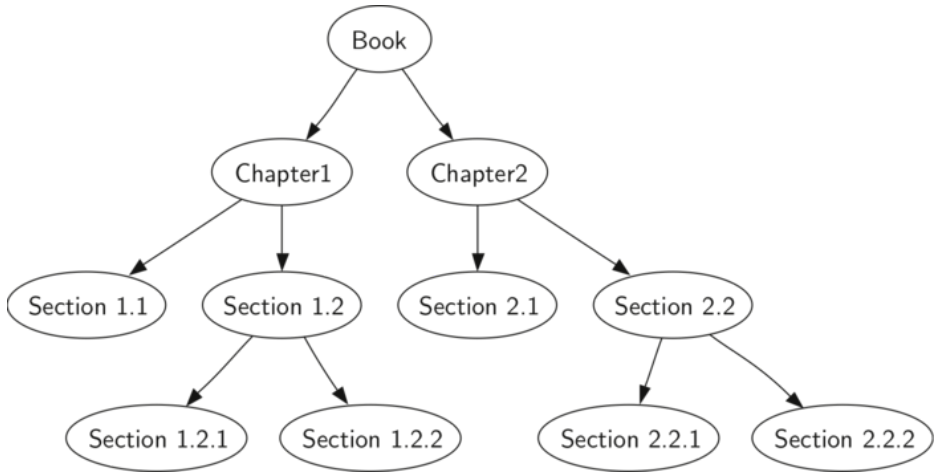
$$u \text{ è } a \textit{ sinistra} \text{ di } v \ (\Rightarrow v \text{ è } a \textit{ destra} \text{ di } u)$$

se u viene prima di v nella visita in preorder.

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

Quale visita è opportuno utilizzare per stampare l'indice di un libro, rappresentato tramite il seguente albero?





La **visita in preorder**! Infatti, la sequenza di visita è

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.

Esempio di algoritmo basato sulla visita in preoder

Vogliamo progettare un algoritmo `allDepths` che: *dato un albero T calcoli la profondità di ogni nodo $v \in T$ e la memorizzi in un campo* v`.depth`.

IDEA: adattare la visita in preorder, definendo la visita di un nodo v in questo modo:

- se v è radice: si imposta la profondità a 0
- se v non è radice: 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 $v \in T$, e u.depth impostato correttamente per

u padre di v

output z.depth impostato correttamente $\forall z \in T_v$

if (T.isRoot(v)) then v.depth ← 0;
else v.depth ← 1 + T.parent(v).depth
} VISITA di v

forall $w \in$ T.children(v) do allDepths(w)

Osservazioni:

* Per impostare il comp depth per tutti i nodi di T invoco allDepths(T, T.root())
* allDepths non restituisce alcun valore (no return) ma modifica i nod dell'albero che si considera come oggetti globali che sopravvivono all'esecuzione dell'algoritmo

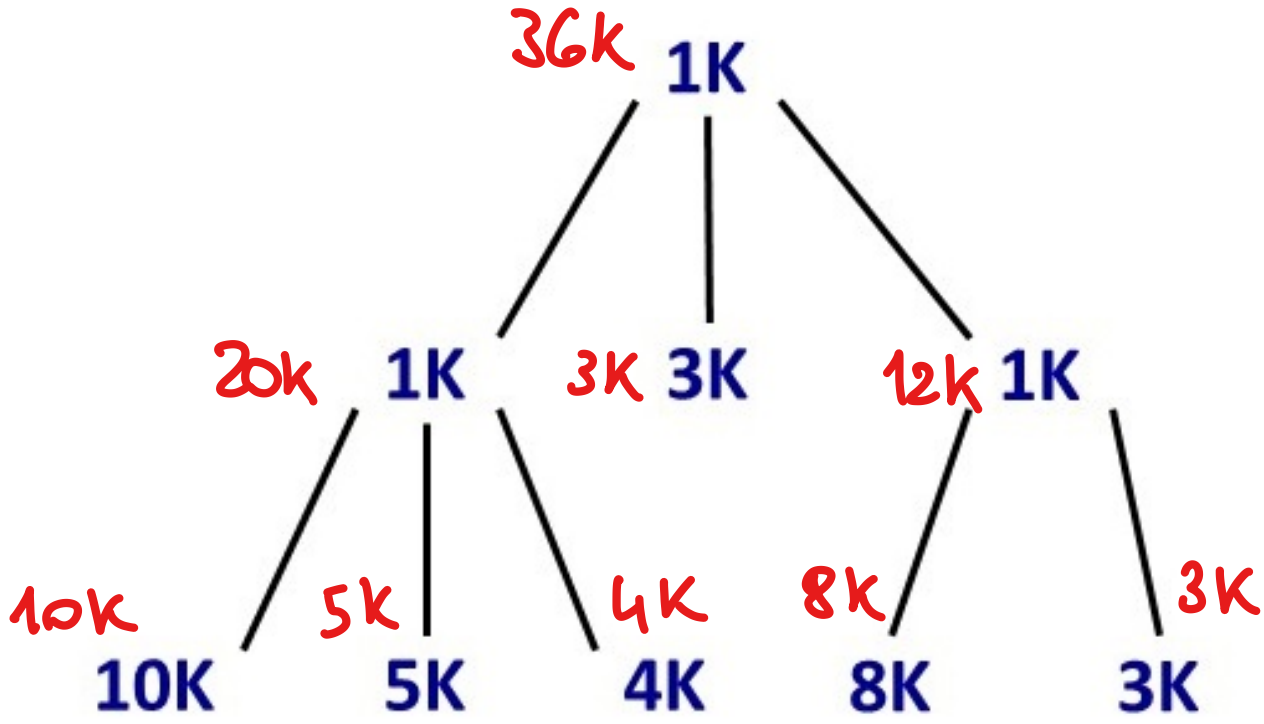
Esempi di algoritmi basati sulla visita in postorder

completamente

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.

Progettare un algoritmo `diskSpace` che: *dato un tale T calcoli, per ogni nodo $v \in T$, lo spazio aggregato occupato dai suoi discendenti e lo memorizzi in un campo v`.aggr-size`*.



- In blue i valori dei campi `loc-size`
- In rosso i valori dei campi `aggr-size` alla fine dell'algoritmo

diskSpace

Algoritmo diskSpace (T, v)

input $v \in T$ e $u.loc\_size$ impostato $\forall u \in T$

output $v.agg\_size$, e $z.agg\_size$ impostato correttamente $\forall z \in T_v$

```
v.agg_size ← v.loc_size;
foreach w ∈ T.children(v) do {
    v.agg_size ← v.agg_size + diskSpace(T, w)
}
return v.agg_size
```

Versione alternativa di diskSpace che non restituisce v.aggr-size in output

input/output: come prima

v.aggr-size ← v.loc_sz;

foreach w ∈ T.children(v) do {

diskSum(T, w)

v.aggr_size ← v.aggr-size + w.aggr-size

}

Osservazioni per la scrittura di algoritmi ricorsivi

- Un algoritmo ricorsivo può utilizzare sia variabili globali che sopravvivono a tutta l'esecuzione dell'algoritmo, e sia variabili locali alle singole invocazioni che rimangono in vita solo durante l'esecuzione della invocazione corrispondente.

- Gli eventuali valori restituiti dalle invocazioni ricorsive (se ce ne sono) devono essere "utilizzati" in qualche modo, altrimenti vanno perduti.

COMPLESSITÀ di preorder(T.root())

Sia n = # nodi di T. Consideriamo l'albero della

ricorsione associato alla esecuzione di preorder(T.root())

* Ha n nodi associati alle invocazioni

ricorsive che sono ESATTAMENTE uno per

ogni nodo di T

* Il costo del nodo associato a preorder(v)

con $v \in T$ generico è $\Theta(t_v + c_v + 1)$

dove t_v = costo di "visita v" e c_v = # figli di v

$\Rightarrow$ Complessità di preorder(T.root()) è

$$\Theta\left(\sum_{v \in T}(t_v + c_v + 1)\right)$$

$$= \Theta\left(\left(\sum_{v \in T} t_v\right) + \sum_{v \in T}(c_v + 1)\right) = \Theta\left(n + \sum_{v \in T} t_v\right)$$

Oss. La stessa complessità la si ha per postorder e inorder (per gli alberi binari)

Dall'esempio discende che la visita consente di enumerare tutti i nodi di T in tempo lineare in n e, se $t_v \in \Theta(1)$ $\forall v$, la complessità totale

Sembla $\Theta(n = |T|)$

COMPLESSITÀ di allDepths(T, T.root())

* Schema della visita in preorder
* $t_v \in \Theta(1) \quad \forall v \in T$

$\Rightarrow$ Complessità $\Theta(n) \quad n = |T|$

COMPLESSITÀ di diskSum(T, T.root())

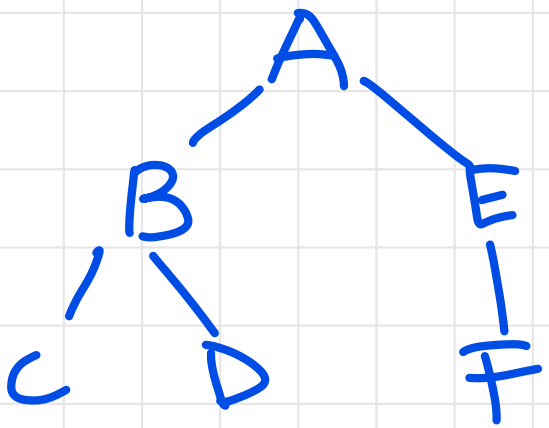
* Schema della visita in postorder
* $t_v \in \Theta(c_v + 1) \quad c_v = \#$ figli di v

$\Rightarrow$ Complessità $\in \Theta\left(n + \sum_{v \in T} t_v\right) = \Theta\left(n + \sum_{v \in T}(c_v + 1)\right) = \Theta(n)$

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. Dire quali delle seguenti sequenze può rappresentare la visita in postorder dello stesso albero (motivando la risposta e disegnando l'albero): BAFECD, CDBFEA, CDAEFB.

* CDBFEA perché la radice è il primo nodo visitato in preorder (⇒ A) e l'ultimo in postorder
* Albero



Esercizio

Si supponga di calcolare l'altezza di un albero T di n nodi invocando l'algoritmo `depth`(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.)

Esercizio (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.

Esercizio

Si consideri un albero T in cui ogni nodo v contiene un valore binario, restitutito da `v.getElement()`, e si dice *alive* se tale valore è 1, e *dead* se esso è 0. Progettare un algoritmo *ricorsivo* `allLiveAncestors` che per ogni nodo $v \in T$ memorizzi in un campo $v.\mathrm{liveAncestors}$ il numero di antenati alive di v, e analizzarne la complessità.

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.

Esercizio

Si consideri un albero binario proprio T dove ciascun nodo v contiene un intero $v.\texttt{val} \geq 0$. Un nodo $v \in T$ si dice *massimale* se $v.\texttt{val} \geq u.\texttt{val}$ per ogni u antenato di v ($\Rightarrow$ la radice è sempre massimale).

1. Progettare in pseudocodice un algoritmo ricorsivo `MaximalSet` che, per ogni nodo v, imposta un flag binario $v.\texttt{maximal}$ a 1 se v è massimale, a 0 altrimenti. Alla fine dell'algoritmo il campo `maximal` di tutti i nodi deve risultare impostato.
2. Analizzare la complessità dell'algoritmo progettato.