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 $\Omega(n^2)$. (È l'algoritmo `heightBad` decritto in [GTG14]. Si veda anche l'esercizio C-8.27 del testo.)

Ricorda che altezza di T = max profondità di una foglia $v \in T$

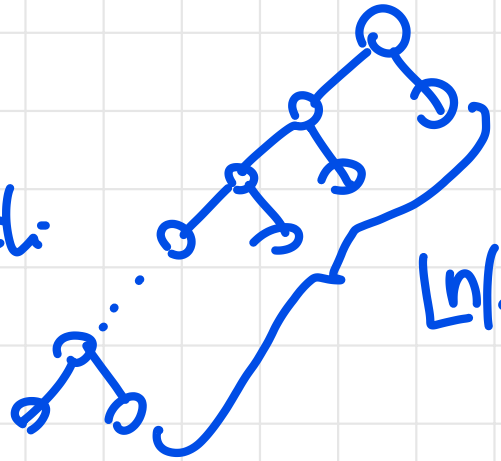
Complessità di depth$(v) \in \Theta(d_v + 1)$

d_v = profondità di v

E' sufficiente dimostrare che invocando depth da ogni foglia $v \in T$ si eseguono $\Omega(n^2)$ operazioni

Istanze cattive

T albero con n nodi



$\lfloor n/2 \rfloor$ nod foglie

le profondità delle foglie sono $\lfloor n/2 \rfloor, \lfloor n/2 \rfloor, \lfloor n/2 \rfloor - 1, \ldots, 1, 0$

$$\Rightarrow \sum_{\substack{v \in T \\ v \text{ foglia}}} (d_v + 1) \in \Omega\left(\sum_{i=0}^{\lfloor n/2 \rfloor}\right) = \Omega(n^2)$$

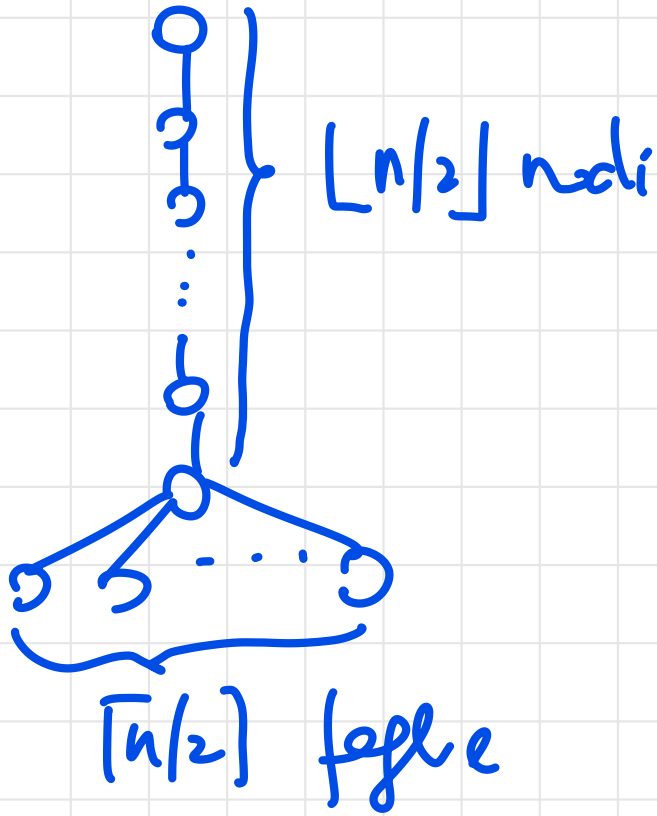
Istanza cattiva dell'architettura

T albero con n nodi

che ha $\lceil n/2 \rceil$ foglie e

profondità $\lfloor n/2 \rfloor$ circa

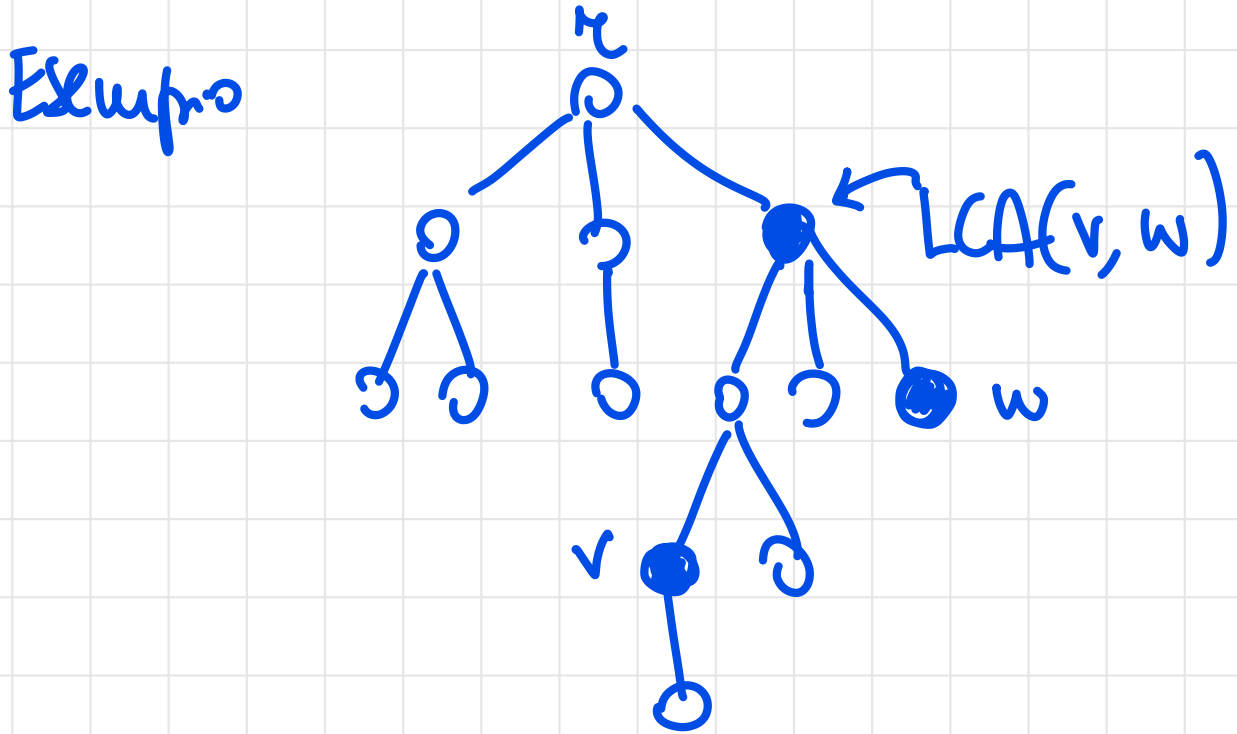
$$\Rightarrow \sum_{\substack{v \in T \\ v \text{ foglia}}} (d_v + 1) \geq \lceil n/2 \rceil \cdot \lfloor n/2 \rfloor \in \Omega(n^2)$$



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 ($\mathrm{LCA}(v, w)$) come l'antenato comune più profondo. Progettare un algoritmo efficiente per trovare $\mathrm{LCA}(v, w)$, analizzandone la complessità.

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



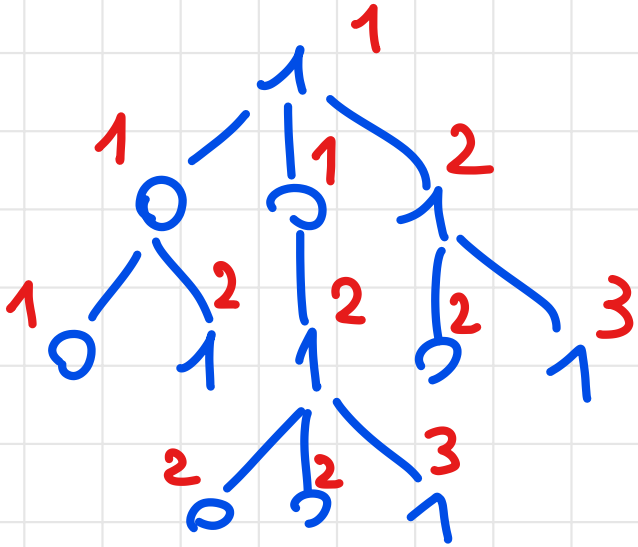
```
Algoritmo LCA(T, v, w)
input  v, w ∈ T
output LCA(v, w)
dv ← T.depth(v) ; dw ← T.depth(w)
 if (dv > dw) then {
    for i ← 1 to dv - dw do  v ← T.parent(v)
 }
 if(dw > dv) then {
    for i ← 1 to dw - dv do  w ← T.parent(w)
 }
```

```
while (v ≠ w) do {
    v ← T.parent(v)
    w ← T.parent(w)
}
return v
```

Complessità $\in \Theta(d_v + d_w)$ ESERCIZIO

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 $v.\mathrm{LA}$ il numero di antenati *alive* di v, e analizzarne la complessità.



in rosso i campi LA

Idea: adattare allDepths

Algoritmo allLiveAncestors(T, v)

input
output } esercizio

if (T.isRoot(v)) then v.LA ← v.getElement()
else v.LA ← v.getElement() + T.parent(v).LA
foreach (w ∈ T.children(v)) do allLiveAncestors(T, w)

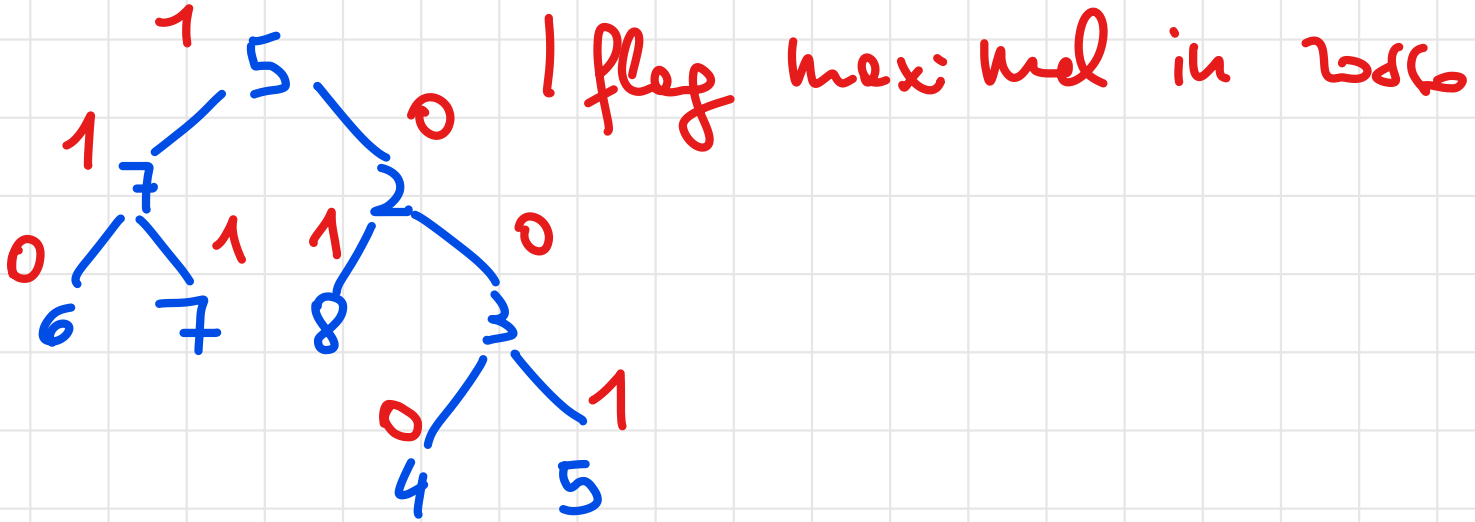
Prima invocazione: v = T.root()

Complessità: esercizio

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.



Algoritmo MexImalSet (T, v, m)

input: $v \in T$, $m = mex\{u.val : u \text{ antenato di } v \text{ in } T, u \neq v\}$

output: $u.maximal$ impostato correttamente per tutti i $u \in T_v$

prima invocazione: $v = T.root()$, $m = -1$

```
if(T.isRoot(v)) then v.maximal ← 1;
else {
   if(v.val ≥ m) then v.maximal ← 1
} else v.maximal ← 0
```

HSITA di v

```
if (T.isInternal(v)) then {
    m' ← max{m, v.val}
    MaximalSet(T, T.left(v), m')
    MaximalSet(T, T.right(v), m')
}
```

Complessità: stessa struttura della visita in preordine, e la visita di un qualsiasi $v \in T$ costa $t_v \in \Theta(1)$

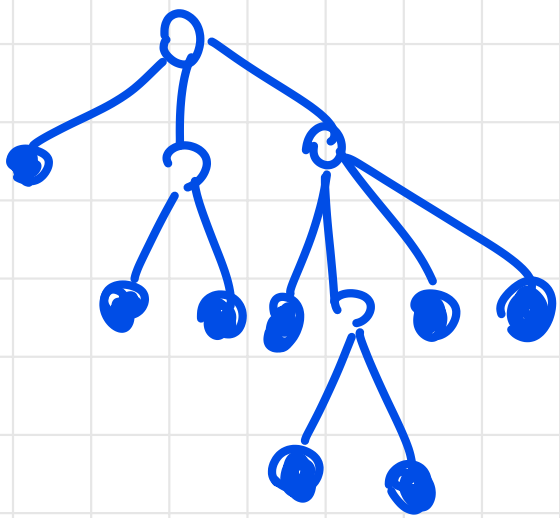
$\Rightarrow$ Complessità $\in \Theta(n)$ $n = \#$ nodi in T

Esercizio

Dimostrare che in un albero non vuoto T con n nodi di cui m foglie, dove ogni nodo interno ha almeno 2 figli, si ha che $m \geq n - m + 1$.

Oss. Se T è binario sappiamo già che $m = n - m + 1$

Esempio



$m = 8$

$n = 12$

$n - m = 4$

$\Rightarrow m \geq n - m + 1$

Mostriamo la prova già vista per alberi binari, e procediamo per induzione su $h \geq 0$

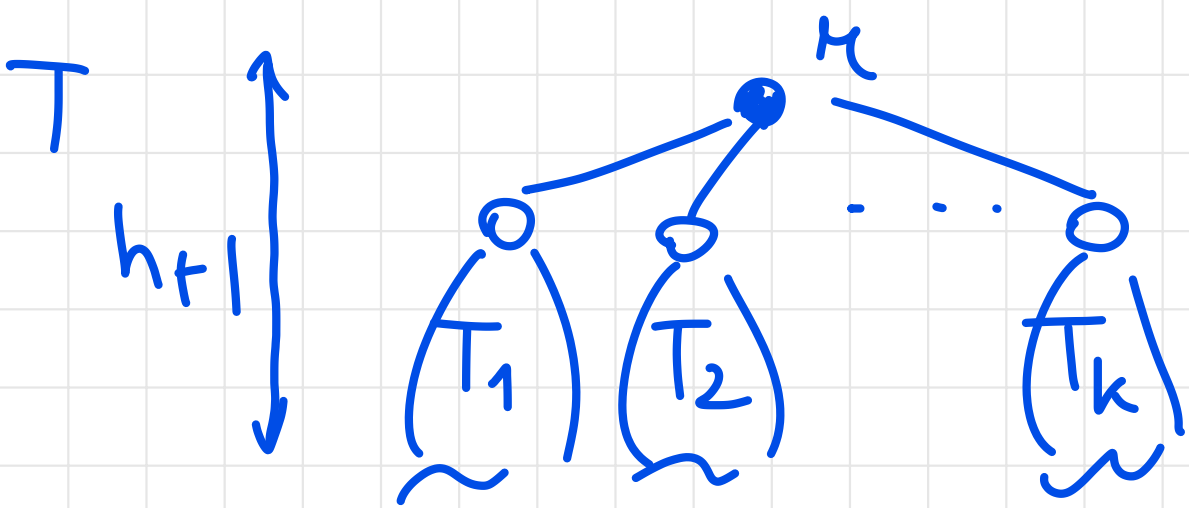
h = altezza di T

BASE: $h = 0 \Rightarrow T \equiv \bullet \Rightarrow m = 1 \quad n = 1 \Rightarrow m \geq n - h + 1$

PASSO INDUTTIVO: Fisso $h \geq 0$

Hp. induttiva: le proprietà vale per tutti gli alberi che soddisfano le ipotesi e di altezza $\leq h$.

Sia T un albero che soddisfa le ipotesi e di altezza $h+1 \geq 1$



$k \geq 2$ figli

h_i = altezza di T_i

$h_i \leq h \ \forall i$

$\Rightarrow \forall T_i$ vale l'ipotesi induttiva

m_i = # foglie di T_i $\quad n_i$ = # nodi in T_i

m = # foglie di T $\quad n$ = # nodi in T

$$m = \sum_{i=1}^{k} m_i \qquad n = \left(\sum_{i=1}^{k} n_i\right) + 1$$

$$m = \sum_{i=1}^{k} m_i$$

$$\geq \sum_{i=1}^{k} (n_i - m_i + 1) \quad \text{par Hyp. Inductive}$$

$$= \left(\sum_{i=1}^{k} n_i\right) - \sum_{i=1}^{k} m_i + k$$

$$= n - m + (k-1)$$

$$\geq n - m + 1 \quad \text{car } k \geq 2$$

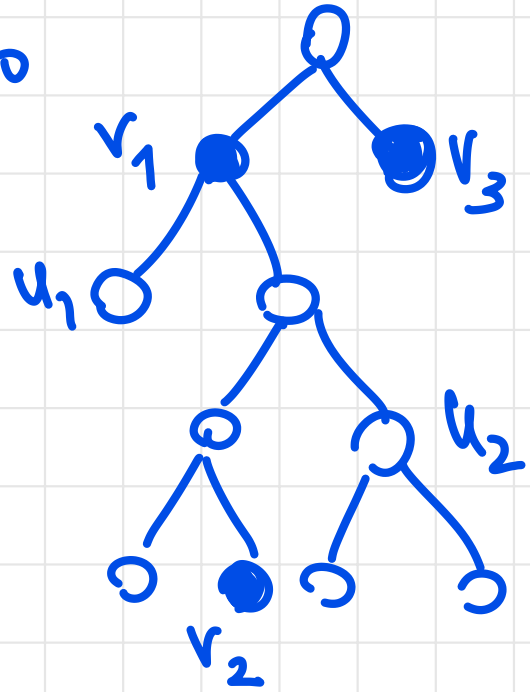
□

Esercizio C-8.41 in [GTG14]

Progettare tre algoritmi *iterativi* `preorderNext(v)`, `inorderNext(v)` e `postorderNext(v)` che dato un nodo v di un albero binario proprio T restituiscano il nodo visitato dopo v nella visita di T rispettivamente in preorder, inorder e postorder, o `null`, se v è l'ultimo nodo visitato, analizzandone la complessità.

Facciamo preorderNext(v)

Esempio



$v_1 \rightarrow u_1$

$v_2 \rightarrow u_2$

$v_3 \rightarrow$ null

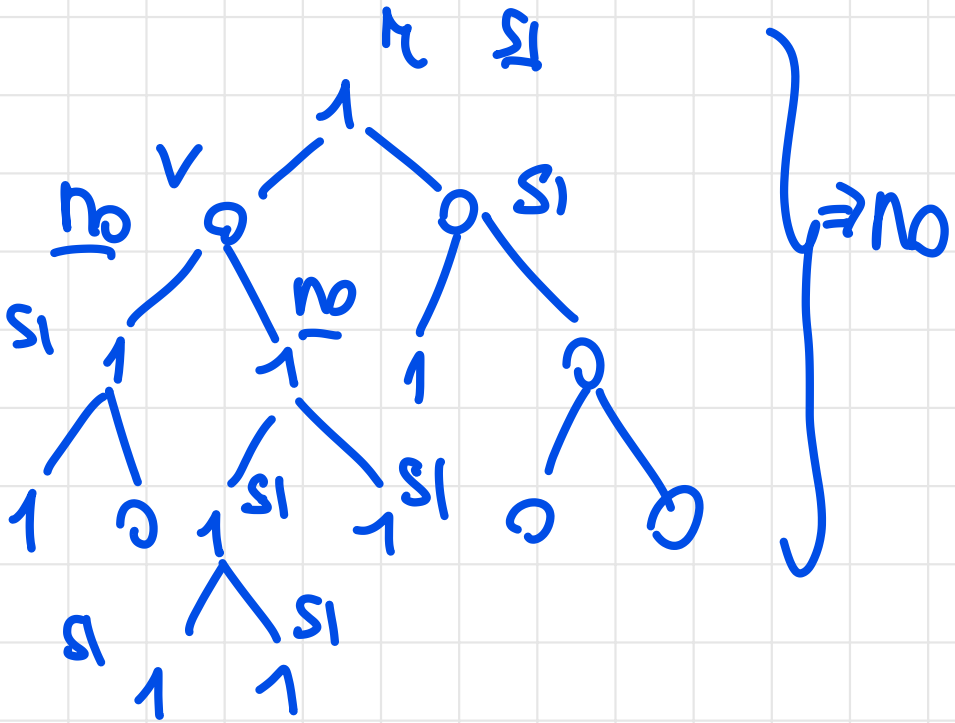
Regole

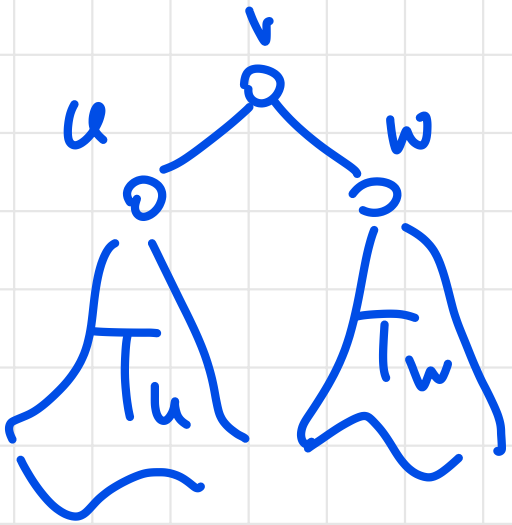
* r interno → figlio sx
* r foglia → fratello dx dell'antenato più profondo che ha un fratello dx se esiste, altrimenti null

Completare per esercizio

Esercizio

Sia T un albero binario proprio dove ogni nodo $v \in T$ contiene un valore binario. Un nodo $v \in T$ si dice *3-balanced*, se $|n_0(v) - n_1(v)| \leq 3$, dove $n_0(v)$ e $n_1(v)$ denotano il numero di 0 ($n_0(v)$) e 1 ($n_1(v)$) nel sottoalbero T_v. Si progetti in pseudocodice un algoritmo ricorsivo `All3Balanced` per determinare se *tutti i nodi di* T *sono 3-balanced*, con complessità lineare nel numero di nodi di T.





Per determinare se tutti i nodi di T_v sono 3-bilanciati devo:

* determinare se tutti i nodi in T_u e T_w lo sono (ricorsivamente)
* determinare se anche v lo è

e quindi calcolare $|n_0(v) - n_1(v)|$ e i valori di $n_0(v)$ e $n_1(v)$ posso ricavarli da quelli dei 2 figli e dal bit in v

(ricorsivamente)

Algoritmo All3Balanced(T, v)

input: $v \in T$ (prima chiamata: $v = T.root()$)

output: (flag, $n_0(v)$, $n_1(v)$) e il flag è true/false se è vero/non vero che tutti i nodi di T_v sono 3-bilanciati

if (v.getElement() = 0) then { $n_v(0) \leftarrow 1$; $n_v(1) \leftarrow 0$ }

else { $n_v(0) \leftarrow 0$; $n_v(1) \leftarrow 1$ }

if (T.isExternal(v)) then return (true, $n_0(v)$, $n_1(v)$)

$(f_L, n_{0,L}, n_{1,L}) \leftarrow$ All3Balanced(T, T.left(v))

$(f_R, n_{0,R}, n_{1,R}) \leftarrow$ All3Balanced(T, T.right(v))

$n_0(v) \leftarrow n_0(v) + n_{0,L} + n_{0,R}$

$n_1(v) \leftarrow n_1(v) + n_{1,L} + n_{1,R}$

if $(|n_0(v) - n_1(v)| \leq 3)$ then return $(f_L \text{ AND } f_R, n_0(v), n_1(v))$

else return $(\text{false}, n_0(v), n_1(v))$

Complessità di `All3Balanced(T, T.root())`:

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

Stesse analisi di `heightSum`

Esercizio

Un *albero di sensori* è un albero binario proprio T i cui nodi rappresentano *sensori*. Ciascun sensore $v \in T$ ha un flag v.`ON` che vale 1 se il sensore è acceso, e 0 se è spento. Si vuole progettare un algoritmo ricorsivo `maxOnHeight` che, dato un albero di sensori T, restituisca la *massima altezza di un sensore acceso*. Se tutti i sensori sono spenti, l'algoritmo restituisce `-1`.

1. Descrivere tramite pseudocodice la generica invocazione di `maxOnHeight`, specificandone con attenzione l'input e l'output.
2. Analizzare la complessità di `maxOnHeight`.

Algoritmo maxOnHeight1(T, v)

input $v \in T$

output (m, h) dove m = massima altezza di un sensore acceso in T_v, h = altezza di v

Completare l'algoritmo e l'accesso per esercizio

Oss. Stessa struttura di heightSum