Dati e Algoritmi: A. Pietracaprina

Alberi Binari

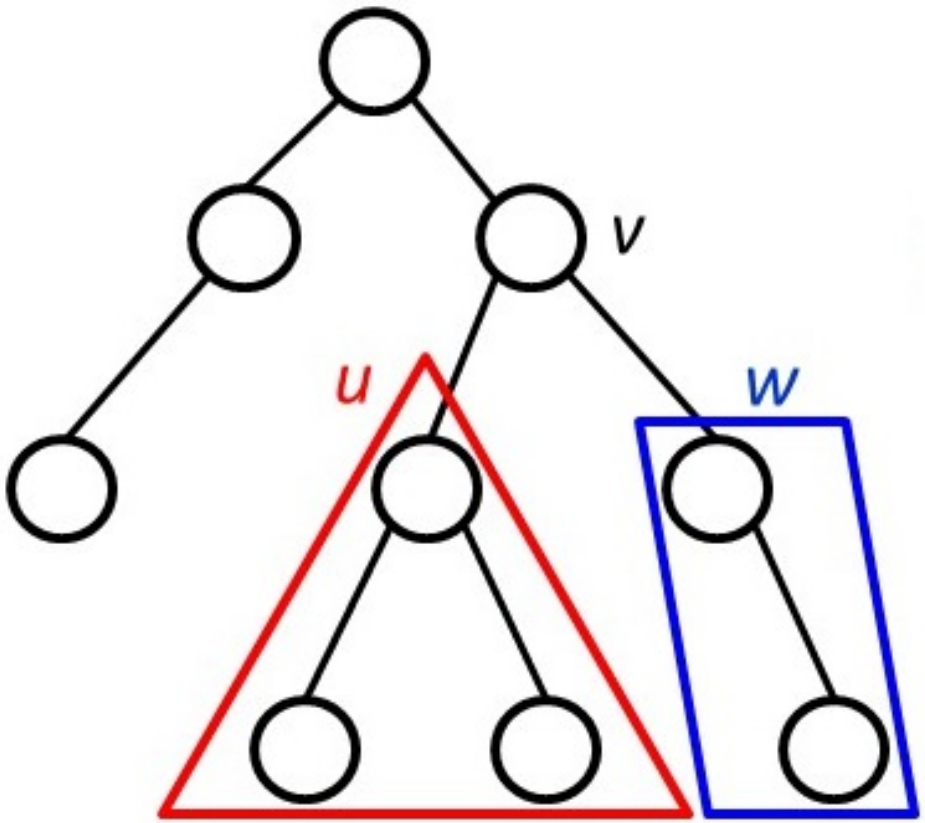
Osservazione: Per *arietà* di un albero si intende il *massimo numero di figli di un nodo interno*

Definizione

Un alberto binario T è un albero ordinato in cui

- ogni nodo interno ha ≤ 2 figli
- ogni nodo non radice è etichettato come figlio sinistro (sx) o destro (dx) di suo padre
- se ci sono entrambi i figli, il figlio sx viene prima del figlio dx nell'ordinamento dei figli di un nodo.

Esempio e Terminologia



u : figlio sx di *v*
w : figlio dx di *v*

sottoalbero sx di *v*

sottoalbero dx di *v*

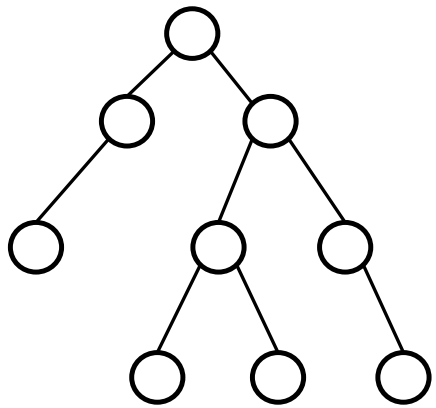
Alberi Binari Propri

Definizione

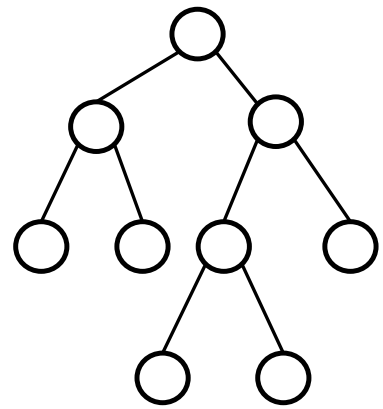
Albero binario proprio T: albero binario tale che:

- ogni nodo interno ha *esattamente* 2 figli

Albero Binario



Albero Binario Proprio



Osservazione

In letteratura gli alberi *propri* (*proper* in inglese) sono anche chiamati *pieni* (*full* in inglese).

Interfaccia BinaryTree

```
public interface BinaryTree<E> extends Tree<E> {
  /** Returns the Position of p's left child (or null if it doesn't exists) */
  Position<E> left(Position<E> p);
  /** Returns the Position of p's right child (or null if it doesn't exists) */
  Position<E> right(Position<E> p);
  /** Returns the Position of p's sibling (or null if no sibling exists) */
  Position<E> sibling(Position<E> p);
}
```

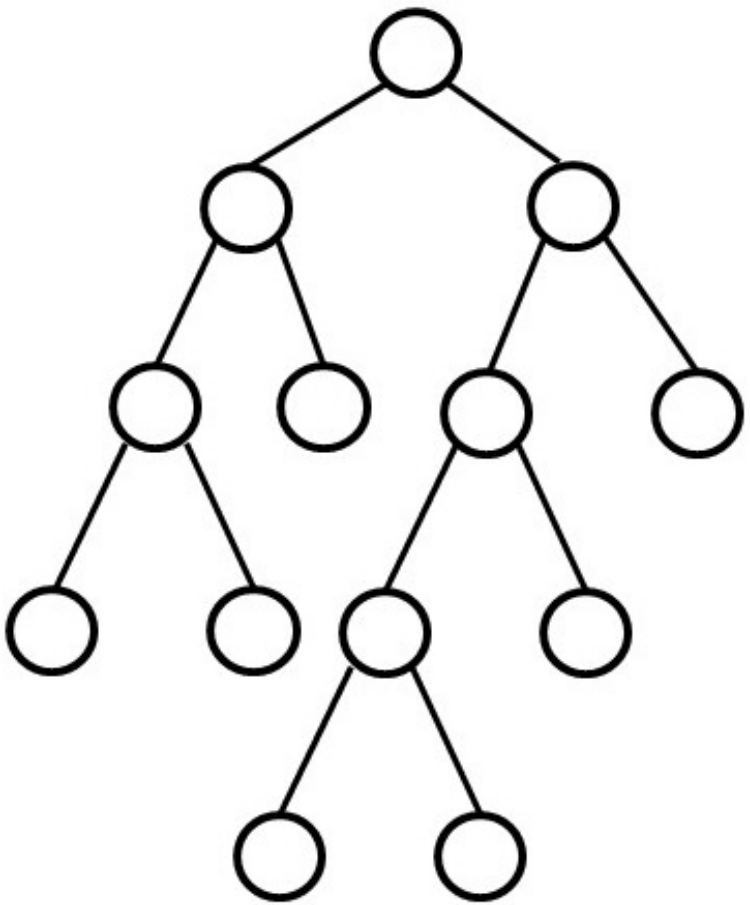
Proprietà di un albero binario proprio T non vuoto

$$
\begin{aligned}
n &= \text{numero di nodi in } T \\
m &= \text{numero di foglie in } T \quad (n_E \text{ in [GTG14]}) \\
n-m &= \text{numero di nodi interni in } T \quad (n_I \text{ in [GTG14]}) \\
h &= \text{altezza di } T
\end{aligned}
$$

Proprietà:

1. $m = n - m + 1$
2. $h + 1 \leq m \leq 2^h$
3. $h \leq n - m \leq 2^h - 1$
4. $2h + 1 \leq n \leq 2^{h+1} - 1$
5. $\log_2(n+1) - 1 \leq h \leq \frac{n-1}{2}$

Esempio



$n = 13$

$m = 7$

$n - m = 6$

$h = 4$

Dimostrazione della Proprietà 1: $m = n - m + 1$

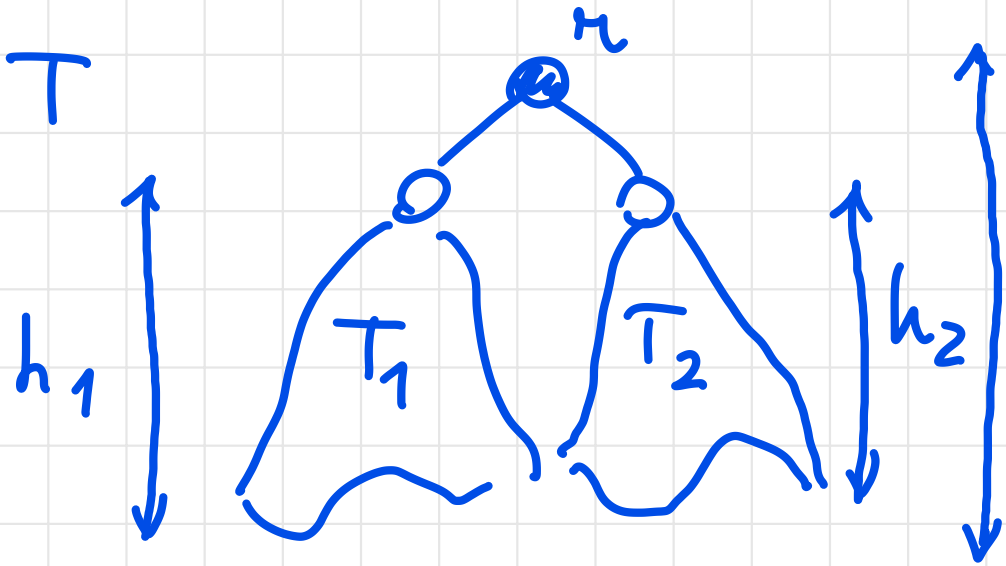
Induzione sull'altezza $h \geq 0$ di T

BASE: $h = 0 \Rightarrow T \equiv \bullet \Rightarrow n = 1 \quad m = 1 \Rightarrow$ Ok

PASSO INDUTTIVO: Fisso $h \geq 0$ arbitrario

Hp. Induttiva: La proprietà 1 vale per tutti gli alberi binari propri di altezza $\leq h$

Sia T un albero binario proprio di altezza $h+1 \geq 1$ e dimostriamo la proprietà per T



$h+1 = 1 + \max\{h_1, h_2\}$

$\Rightarrow h_1, h_2 \leq h$

$\Rightarrow$ Per T_1, T_2 vale l'Hp. Induttiva

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

$m_i = \#$ foglie di T_i $\quad n_i = \#$ nodi di T_i $\quad i = 1,2$

$m = m_1 + m_2$ $\quad n = n_1 + n_2 + 1$

$$
\begin{aligned}
n &= n_1 + n_2 \\
&= (n_1 - m_1 + 1) + (n_2 - m_2 + 1) \quad \text{per Hp. Induttiva} \\
&= (n_1 + n_2 + 1) - (m_1 + m_2) + 1 \\
&= n - m + 1 \qquad \square
\end{aligned}
$$

Oss. In un albero binario proprio T, dato che il numero di foglie è uguale al numero di nodi interni + 1, il numero totale di nodi è dispari

Dimostrazione della Proprietà 2: $h + 1 \leq m \leq 2^h$

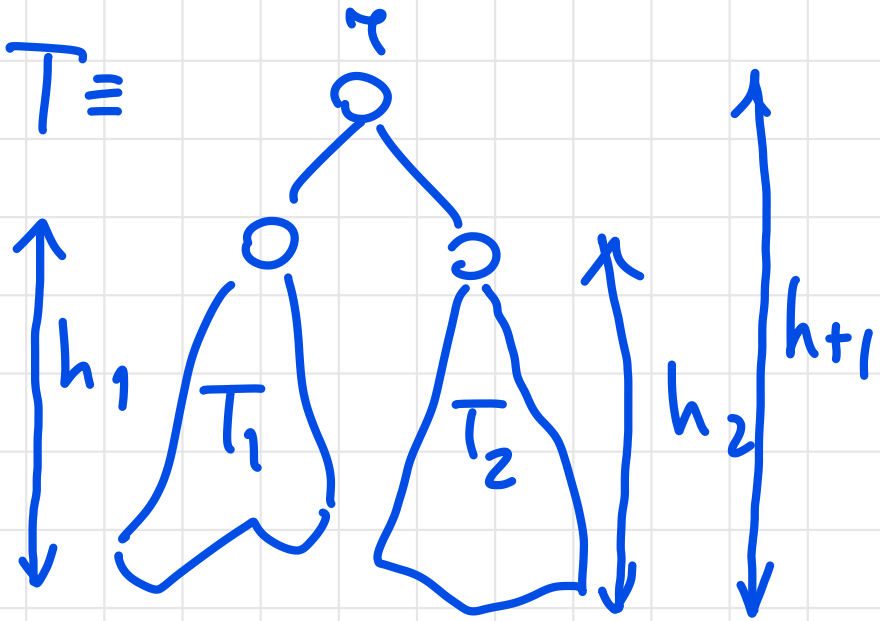
Dimostriamo che $m \leq 2^h$ per induzione su $h \geq 0$

BASE: $h = 0 \Rightarrow T \equiv \bullet \Rightarrow n = 1 \Rightarrow$ ok

PASSO INDUTTIVO: Fisso $h \geq 0$ arbitrario

Hp. Induttiva: la proprietà vale per tutti gli alberi binari propri di altezza $\leq h$

Sia T un albero binario proprio di altezza $h+1 \geq 1$



Come possiamo dedurre che $h_1, h_2 \leq h$

$\Rightarrow$ l'Hp Induttiva vale per T_1 e T_2

n = # foglie di T

n_i = # foglie di T_i $\quad i = 1,2$

$n = n_1 + n_2$ e devo dimostrare che $n \leq 2^{h+1}$

$$n = n_1 + n_2$$
$$\leq 2^{h_1} + 2^{h_2} \quad \text{per ipotesi induttiva}$$
$$\leq 2^h + 2^h = 2^{h+1}$$

□

Esercizio: Dimostrare che $n \geq h+1$ per induzione su $h \geq 0$

Dimostrazione delle Proprietà 3,4,5

1. $m = n - m + 1$
2. $h + 1 \leq m \leq 2^h$
3. $h \leq n - m \leq 2^h - 1$
4. $2h + 1 \leq n \leq 2^{h+1} - 1$
5. $\log_2(n+1) - 1 \leq h \leq \frac{n-1}{2}$

P3: Sostituisco in P2 m con $n-m+1$ (da P1)

$\Rightarrow h+1 \leq n-m+1 \leq 2^h \Rightarrow h \leq n-m \leq 2^h - 1$

P4: P2 + P3

P5: Deriva da P4 risolvendo rispetto ad h

Da P4:

* $2h+1 \leq n \Rightarrow h \leq (n-1)/2$

* $2^{h+1} - 1 \geq n \Rightarrow 2^{h+1} \geq n+1$

$\Rightarrow h+1 \geq \log_2(n+1)$

$\Rightarrow h \geq \log_2(n+1) - 1$

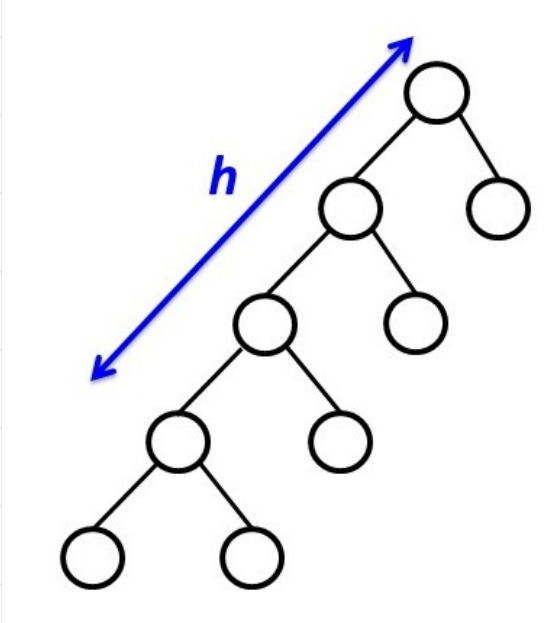
OSSERVAZIONE: la P5 implica che in un albero binario proprio con n nod l'altezza è compresa tra $\Omega(\log n)$ e $O(n)$

Osservazioni

In [GTG14]:

- Proprietà 1: Proposition 8.8 $\Rightarrow$ Esercizio R-8.8
- Proprietà 2, 3, 4 e 5: Proposition 8.7 $\Rightarrow$ Esercizio R-8.7
  (Nel testo la proposizione enuncia proprietà analoghe anche per alberi non propri.)

Alberi Binari Propri Estremi



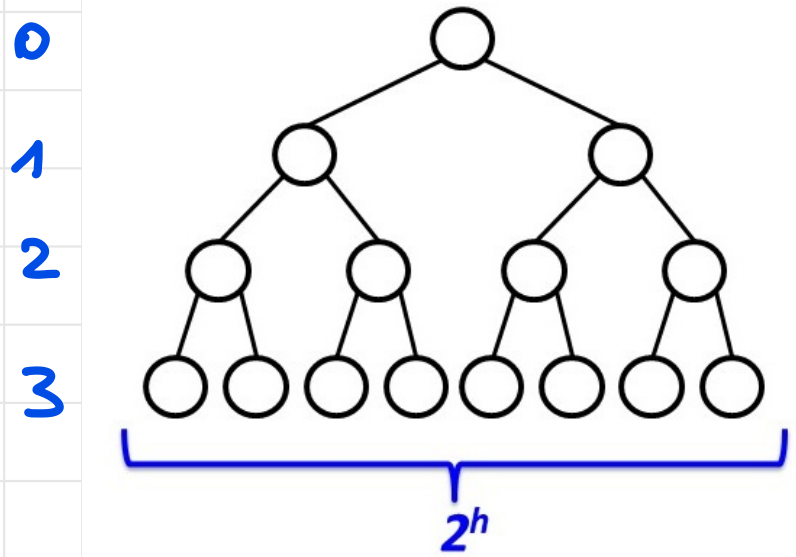
$m = h + 1$

$n = 2h + 1$

Esempio per le parti SX di P2, P3, P4, e dalle parte dx di P5

ESEMPIO DI ALBERO MOLTO SBILANCIATO (SKEWED)

Livello



2^i nodi al livello i, $0 \le i \le h$

$\Rightarrow n = 2^h$ (= nodi al livello h)

$$\Rightarrow n = \sum_{i=0}^{h} 2^i = \frac{2^{h+1}-1}{2-1} = 2^{h+1} - 1$$

Esempio per la parte dx di P2, P3, P4, e per la parte sx di P5

ESEMPIO DI ALBERO PERFETTAMENTE BILANCIATO

IMPORTANTE

In assenza di altre ipotesi,
il migliore upper bound all'altezza di un albero binario con n nodi è

$$O(n)$$

e non $O(\log n)$

Visite di Alberi Binari

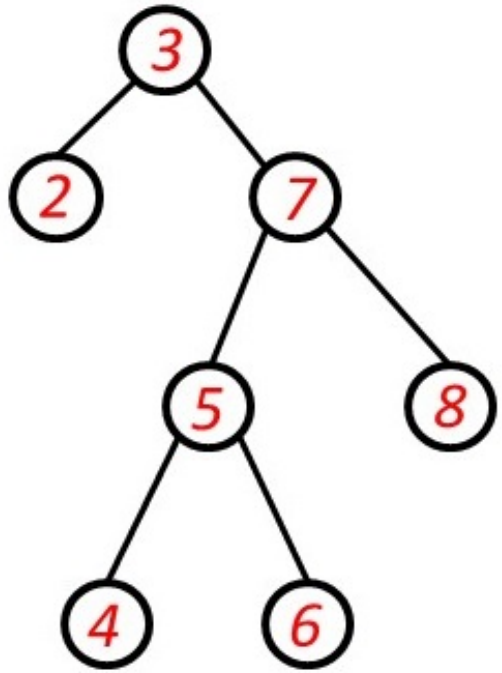
Oltre alle visite in preorder e postorder, per gli alberi binari si definisce anche la *visita inorder*, basata sulla regola:

prima (ricorsivamente) il sottoalbero sx,
poi il padre,
poi (ricorsivamente) il sottoalbero dx.

Algoritmo `inorder(`v`)`
Input: $v \in T$
Output: visita inorder di T_v
if (`T.left(`v`)` $\neq$ `null`) **then** `inorder(`T`.left(`v`))`;
visita v;
if (`T.right(`v`)` $\neq$ `null`) **then** `inorder(`T`.right(`v`))`;

Chiamata iniziale: inorder(T.*root*()).

Esempio



Sequenza inorder:

Osservazione: È un esempio di albero binario di ricerca, che studieremo più avanti, dove la visita inorder tocca gli elementi presenti in ordine crescente di valore.

Complessità di inorder(T.*root*())

Complessità $\in \Theta(n + \sum_{v \in T} t_v)$ dove $n = |T|$ e

t_v = costo della visita di v

Stessa analisi di prima

Applicazioni delle visite: espressioni aritmetiche

Definizione (Parse Tree)

Il Parse Tree T associato a una espressione aritmetica E (con operatori solo binari) è un albero binario proprio i cui nodi foglia contengono le costanti/variabili di E e i nodi interni contengono gli operatori di E, in modo tale che:

- Se $E = a$, con a costante/variabile, allora T è costituito da un'unica foglia contenente a
- Se $E = (E_1\ \mathrm{Op}\ E_2)$, la radice di T contiene Op e ha come sottoalbero sx (risp., dx) il Parse Tree associato a E_1 (risp., E_2).

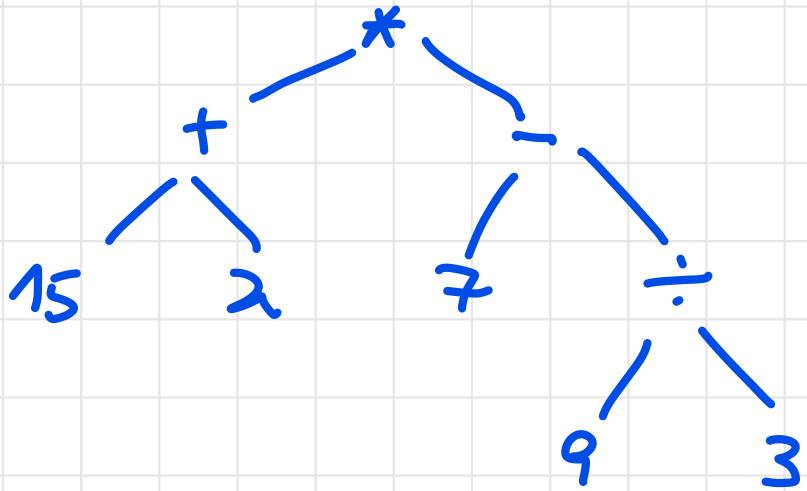
Esempio

$E = ((15 + 2) * (7 - (9 \div 3)))$ NOTAZIONE INFISSA

$= 15\ 2\ +\ 7\ 9\ 3\ \div\ -\ *$ NOTAZIONE POSTFISSA

o POLACCA INVERSA

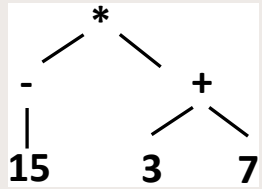
PARSE TREE PER E



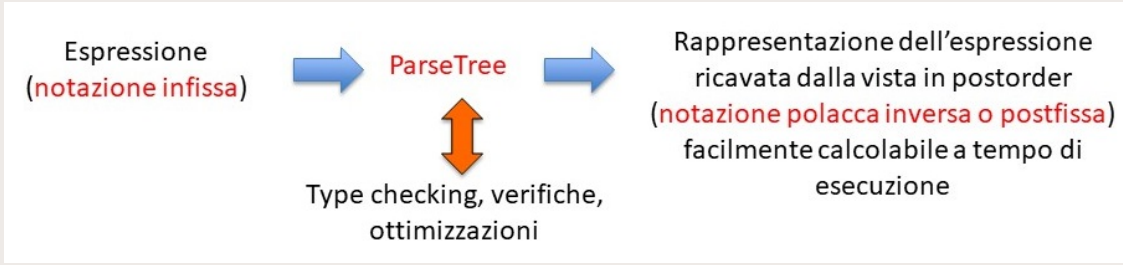
Osservazione: le precedenze sono implicite nella struttura dell'albero

Osservazioni

1. Se si ammettono operatori unari, l'albero non è più proprio. Ad es.: $-15 * (3 + 7)$



2. Nei compilatori



Nelle slide successive vedremo come a partire dal Parse Tree T di una espressione E si possono generare le rappresentazioni di E in notazione postfissa e infissa, usando le visite rispettivamente in postorder e inorder.

Sia E_v la espressione associata al sottoalbero T_v, con v nodo di T

Generazione della espressione in notazione infissa

Idea: si utilizza lo schema della visita inorder

E_v = espressione rappresentata da T_v

Algoritmo `infix`(T,v,L)
Input: Parse Tree T for E, $v \in T$, Lista L
Output: Aggiunta a L di E_v in notazione infissa

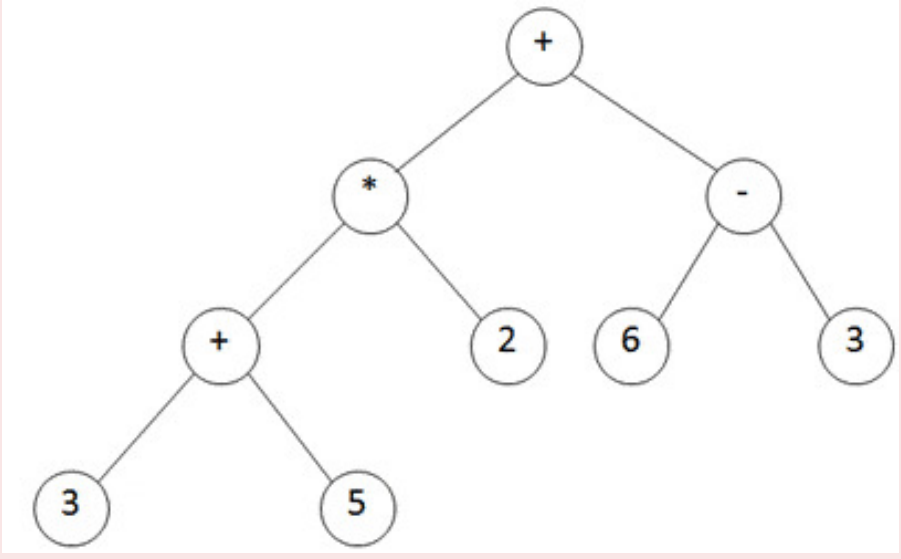
```
if (T.isExternal(v)) then L.addLast(v.getElement());
else
    L.addlast('(');
    infix(T, T.left(v), L);
    L.addlast(v.getElement());
    infix(T, T.right(v), L);
    L.addlast(')');
```

Chiamata iniziale: infix(T, $T.root()$, L) con $L = \emptyset$.

Oss. T e L sono variabili globali

Esercizio

Disegnare l'albero della ricorsione relativo all'esecuzione di `infix` sul seguente Parse Tree, determinando la lista L di uscita.



Complessità di infix(T, $T.root()$, L)

Stessa struttura della visita inorder e la complessità è $\Theta(n + \sum_{v \in T} t_v)$ dove

$n = |T|$

t_v = costo della visita di v che in questo caso è $\Theta(1)$

$\Rightarrow$ complessità $\in \Theta(n)$

Generazione della espressione in notazione postfissa

Idea: si utilizza lo schema della visita in postorder

Algoritmo `postfix`(T,v,L)
Input: Parse Tree T for E, $v \in T$, Lista L
Output: Aggiunta a L di E_v in notazione postfissa

if (`T.isExternal(v)`) **then** L.addLast(v.getElement());
else
    postfix(T,T.left(v),L);
    postfix(T,T.right(v),L);
    L.addlast(v.getElement());

visita di $v \in \Theta(1)$

Chiamata iniziale: postfix(T, $T.root()$, L) con $L = \emptyset$.

Esercizio: applicare l'algoritmo all'esempio della slide precedente

Complessità di postfix(T, $T.root()$, L)

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

Stessa analisi di infix

Esercizio

Progettare e analizzare un algoritmo che dato un Parse Tree T restituisce il valore della espressione E associata a T, assumendo che i valori di costanti e variabili siano noti.

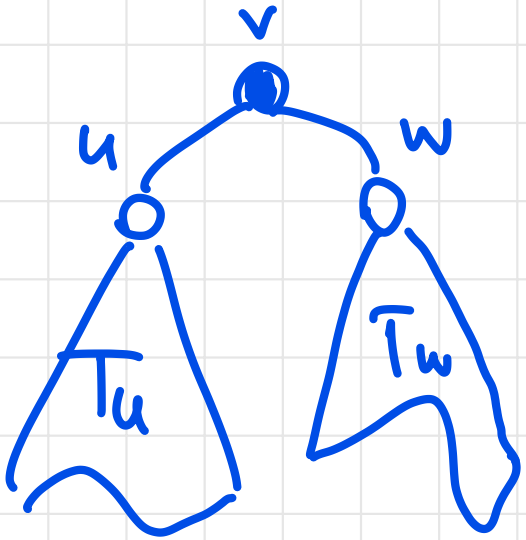
Esercizio

Progettare e analizzare un algoritmo che data una lista che rappresenta una espressione E in notazione postfissa, restituisce il valore di E, assumendo che i valori di costanti e variabili siano noti.

Esercizio

Si vuole progettare un algoritmo *ricorsivo* `heightSum` per calcolare la somma delle altezze di tutti i nodi di un albero binario proprio T. Detta `heightSum(T,v)` la generica invocazione su un nodo $v \in T$, per risolvere il problema si eseguirà `heightSum(T,T.root())`.

1. Descrivere tramite pseudocodice `heightSum(T,v)`, specificandone con attenzione l'input e l'output.
2. Analizzare la complessità di `heightSum(T,T.root())` in funzione del numero n di nodi in T.



heightSum(T, v) =
heightSum(T, u) +
heightSum(T, w) +
altezza di v

Segue lo schema di visita in postorder

PROBLEMA: Per calcolare la altezza di v posso invocare height(v) ma in questo modo ricalcolo tutte le altezze dei nodi di T_v che ho già calcolato nelle chiamate ricorsive ⇒ inefficiente

SOLUZIONE: Calcolare ricorsivamente T_u e T_w fa la somma delle altezze dei loro nodi, e fa le loro altezze, cioè l'altezza di u e w

Algoritmo heightSum (T, v)

input $v \in T$

output (s, h) : s = somma delle altezze dei nodi di T_v, h = altezza di v

PRIMA CHIAMATA: $v = T.root()$

if (T.isExternal(v)) then return (0,0)

$(s_L, h_L) \leftarrow$ heightSum(T, T.left(v))

$(s_R, h_R) \leftarrow$ heightSum(T, T.right(v))

$h \leftarrow \max\{h_L, h_R\} + 1$

$s \leftarrow s_L + s_R + h$

return (s,h)

→ visita di v

Complessità:

* Stessa struttura di postorder
* Costo della visita di v è $t_v \in \Theta(1)$

$\Rightarrow$ Complessità di heightSum(T, T.root()) è $\Theta(n)$ con $n = |T|$ (# nod.)

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 $m \geq n - m + 1$.

Esercizio

Dimostrare che per ogni $n > 0$ dispari, esiste un albero binario proprio con n nodi e altezza $h \in \Theta\left(\sqrt{n}\right)$.

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.

Esercizio

Si vuole progettare un algoritmo *ricorsivo* `deepLeaf` per trovare la foglia più profonda in un albero binario proprio T.

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

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.

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`.

Riepilogo (alberi generali e binari)

- Definizioni: albero, albero binario (proprio), antenati, discendenti, sottoalbero, profondità di un nodo, altezza di un nodo e di un albero.
- Relazione tra altezza di un albero e profondità delle foglie
- Algoritmi per il calcolo della profondità/altezza di un nodo e dell'altezza di un albero
- Relazioni tra numero di nodi interni, foglie e altezza in un albero binario proprio
- Visite: preorder, postorder, inorder e loro applicazioni
- Algoritmi di visita come template generali