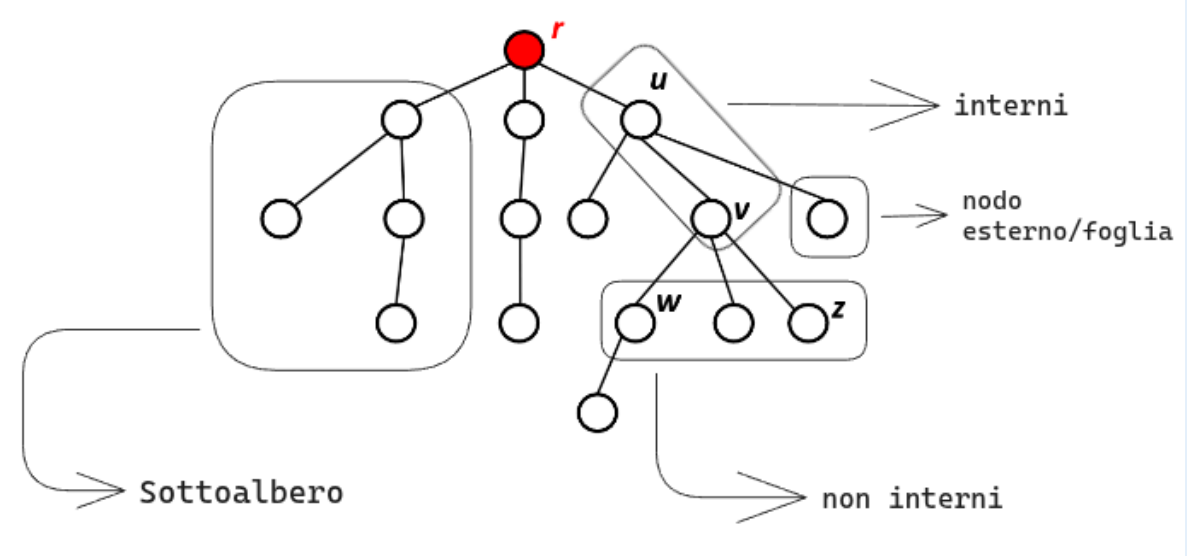
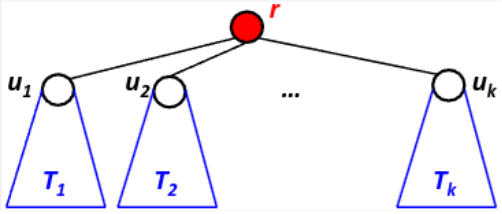
Lezione_10_DeA

Terminologia

- x è *antenato* di y se $x = y$ oppure x è antenato del padre di y;
- x è *discendente* di y se y è antenato di x;
- I *nodi interni* sono quei nodi con almeno 1 figlio;
- I *nodi esterni* (o *foglie*) sono i nodi senza figli;
- T_v è un albero formato da tutti i discendenti di v (quindi include v);
- 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.



Si può definire ricorsivamente cos'è un albero radicato.

> Un *albero radicato* T è una collezione di nodi che, se non è vuota, risulta partizionata in questo modo:
>
> $$T = \{r\} \cup T_1 \cup T_2 \cup \ldots \cup T_k$$
>
> 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 ($\implies T_i \equiv T_{u_i}$).
>
> 
>

Chiaramente, le due definizione di albero radicato (ricorsiva e non) sono equivalenti.

> Si definisce la *prodondità* di un nodo v in un albero T in due modi alternativi:
>
> 1. $\text{depth}_T(v) = |\text{antenati}(v)| - 1$;
> 2. - Se $v = r$ radice $\implies \text{depth}_T(v) = 0$;
>   - Altrimenti $\text{depth}_T(v) = 1 + \text{depth}_T(\text{padre}(v))$.

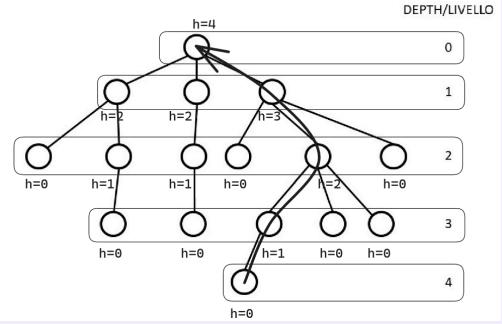
> Il *livello* è l'insieme dei nodi a profondità i ($froalli \geq 0$).

> L'*altezza* di un nodo v in un albero T ($\text{height}_T(v)$) si definisce come:
>
> - Se v è foglia $\implies \text{height}_T(v) = 0$;
> - Altrimenti $\text{height}_T(v) = 1 + \max_{w: w \text{ figlio di } v}(\text{height}_T(w))$.

> L'*altezza di un albero* T si definisce $\text{height}(T) = \text{height}_T(r)$, con r radice di T.

Proposizione

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

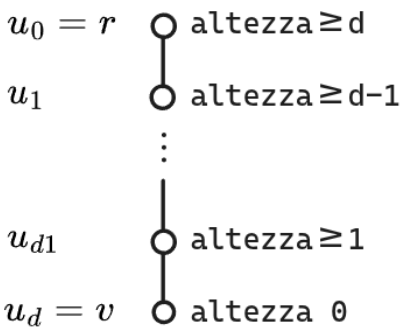


4.1.1.1. Dimostrazione

Sia h l'altezza dell'albero e $d = \max_{v \in T: v \text{ foglia}}(\text{depth}_T(v))$. Proviamo $h \geq d$ e $h \leq d$.

4.1.1.1.1. $h \geq d$

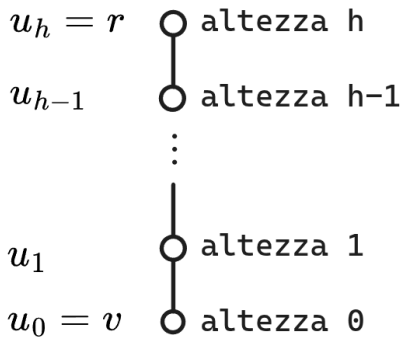
Sia v una foglia a profondità d e sia r la radice di T. allora esiste un percorso da v a r.



È immediato vedere che
$\text{height}_T(u_i) \geq d - i \forall d \geq i \geq 0 \implies \text{height}_T(r) \geq d \implies h \geq d$.

4.1.1.1.2. $h \leq d$

Per assurdo, se $h \geq d$ deve esistere un percorso in T dalla radice r a una foglia v.



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

□

4.1.2. Interfacce

4.1.2.1. Iterator e Iterable

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

> `Iterator` è un "cursore" che permette di enumerare (scan) gli elementi di una collezione.

```
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();
}
```

> `Iterable` è una collezione che rende disponibile un iteratore ai suoi elementi.

```
public interface Iterable<E> {
     /** Returns an iterator of the collection */
     Iterator<E> iterator()
}
```

4.1.2.1. Tree

L'interfaccia `Tree` rappresenta l'implementazione tipica di un albero.
Notare che il puntatore alla radice è l'unico punto di accesso e che ogni nodo è un oggetto a sé stante (ad esempio di una classe che implementa `Position`) e offre metodi per accedere al padre e ai figli.

```
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>` (dove `E` rappresenta il tipo dei dati contenuti nei nodi).
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 (quindi ogni figlio è enumerato in tempo costante).

4.1.3. Calcolo della profondità di un nodo (algoritmo ricorsivo)

Algoritmo `depth(v)`
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

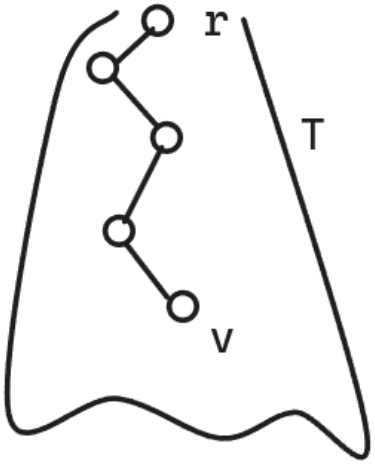
Come nel libro di testo, definiamo gli algoritmi di base per gli alberi come metodi di una classe astratta che implementa l'interfaccia `Tree`, non specificando l'albero T come parametro in quanto esso è associato implicitamente all'istanza (`this`) da cui si invoca il metodo.

4.1.3.1. Complessità di `depth`

Studiamo l'albero della ricorsione associato a `depth(v)`, $v \in T$.
Se d_v è la profondità di v:

- Vi sono $d_v + 1$ invocazioni ricorsive di `depth`, una per ogni antenato di v;
- Avvengono $\Theta(1)$ operazioni in ciascuna invocazione ricorsiva.
  $\implies$ Vi sono $d_v + 1$ nodi dell'albero della ricorsione di costo $\Theta(1)$ ciascuno.

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



> **Osservazione**
>
> Il "$+1$" è giustificato dal fatto che se $d_v = 0$ (ovvero v è la radice), comunque `depth(v)` richiede $\Theta(1)$ operazioni; *tuttavia* per semplificare la notazione scriveremo $\Theta(d_v)$ intendendo $\Theta(d_v + 1)$.

4.1.3.2. Diversa taglia dell'istanza

Se volessi esprimere la complessità in funzione di $n = |T|$?

Usando la profondità d come taglia dell'istanza, le possibili istanze di taglia sono tutti i nodi di tutti i possibili alberi T (di qualsiasi cardinalità $|T|$) $\implies$ La complessità al caso pessimo è $\Theta(d)$.

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 $|T| = n$ $\implies$ La complessità al caso pessimo è $\Theta(n)$.

4.1.3.2.1. Dimostrazione di complessità

L'upper bound è $O(n)$ perché in un albero con n nodi ogni nodo v ha profondità $len \implies$ `depth(v)` richiede $O(n)$ operazioni.
Il lower bound è $\Omega(n)$ perché in un albero che è una catena di n nodi, l'ultimo nodo della catena è una foglia v di profondità $n-1$, e quindi `depth(v)` richiede $\Omega(n)$ operazioni.

4.1.3.3. Versione iterativa

Algoritmo `depthITER(v)`
Input: $v \in T$.
Output: profondità di v in T.

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

La sua complessità è $\Theta(d_v)$.

4.1.4. Calcolo dell'altezza di un nodo

Algoritmo `height(v)`
Input: $v \in T$.
Output: altezza di v in T.

```
h <- 0;
foreach w ∈ T.children(v) do{
     h <- max{h, 1 + height(w)};
}
return h;
```

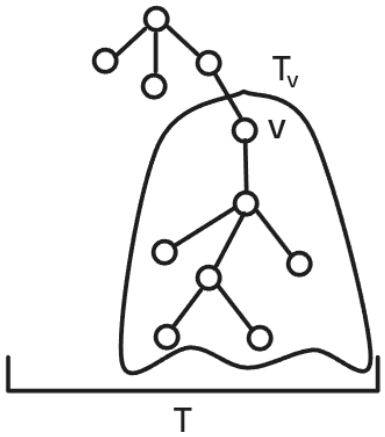
Tecnicamente, `T.children(v)` è una collezione "Iterable" che contiene i figli di v. Con "`foreach w ∈ T-children(v)`" si enumerano i figli di v, ciascuno dei quali viene generato in tempo $\Theta(1)$.

4.1.4.1. Complessità di `height`

Studiamo l'albero della ricorsione associato a `height(v)` con $v \in T$:

- Ha un nodo (cioè un'invocazione ricorsiva) per ogni $u \in T_v$;
- Il costo associato al nodo $u \in T_v$ è $\Theta(c_u + 1)$, dove c_u è il numero di figli di u

  $\implies$ La complessità di `height(v)` $\in \Theta\left(\sum_{u \in T_v}(c_u + 1)\right)$



Sia n_v il numero di nodi in T_v (quindi il numero di discendenti di v). Riscriviamo $\sum_{u \in T_v}(c_u + 1)$ in funzione di n_v:
$\sum_{u \in T_v}(c_u + 1) = \left(\sum_{u \in T_v} c_u\right) + \sum_{u \in T_v} 1 = \left(\sum_{u \in T_v} c_u\right) + n_v$.

> **Proposizione (8.4 del libro di testo)**
>
> $\sum_{u \in T_v} c_u = n_v - 1$

$\implies \sum_{u \in T_v}(c_u + 1) = 2n_v - 1$
$\implies$ La complessità di `height(v)` è $\Theta(n_v)$.

4.1.4.1.1. Dimostrazione della proposizione

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 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.