Lezione_11_DeA

4.1.5. Visite di alberi

> La *visita di un albero* T è la scansione sistematica di 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 di opportune variabili associate ai nodi.

Per gli alberi generali studieremo la visita in preorder e in postorder.

4.1.5.1. Preorder

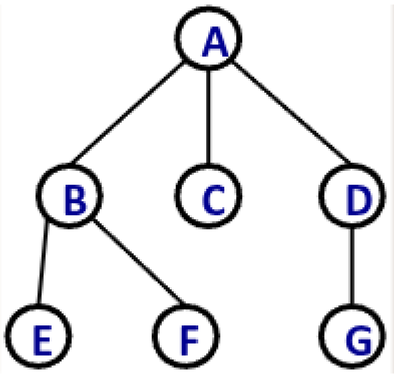
> Una visita in *preorder* visita *prima il padre e poi (ricorsivamente) i sottoalberi radicati nei figli*.

Con questa modalità, le operazioni svolte per un nodo possono dipendere da quelle svolte per i suoi antenati.

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

```
visita v;
foreach w ∈ T.cildren(v) do{
    preorder(w);
}
```

La *chiamata iniziale* da effettuare per visitare tutto T è `preorder(T.root())`.
Nel *caso base* v è una foglia (il ciclo `foreach` non esegue istruzioni, infatti v non ha figli).



Assumendo che `children()` esamini i figli da sinistra a destra, l'ordine della visita in preorder per l'albero sovrastante è A, B, E, F, C, D, G.

4.1.5.2. Postorder

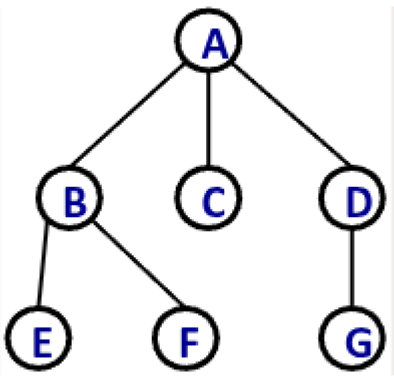
> Una visita in *postorder* visita *prima (ricorsivamente) i sottoalberi radicati nei figli, poi il padre*.

Con questa modalità, le operazioni svolte per un nodo possono dipendere da quelle svolte per i suooi discendenti.

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

```
foreach w ∈ T.children(v) do{
    postorder(w);
}
visita v;
```

La *chiamata iniziale* da effettuare per visitare tutto T è `postorder(T.root())`.
Nel *caso base* v è una foglia (il ciclo `foreach` non esegue istruzioni, infatti v non ha figli).



Assumendo che `children()` esamini i figli da sinistra a destra, l'ordine della visita in preorder per l'albero sovrastante è E, F, B, C, G, D, A.

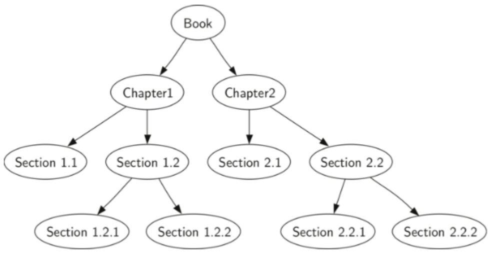
> Sia T un albero ordinato e siano $u, v \in T$ due nodi allo stesso livello.
> Diciamo che u è *a sinistra* di v (e quindi v è a destra di u) se u viene prima di v nella visita in *preorder*.

Osservazione

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

4.1.5.3. Esempi

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



Quella corretta è 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".

Un esempio analogo è rappresentato dalla struttura di un file system come sequenza di cartelle e file.

4.1.5.3.1. Esempio di algoritmo basato sulla visita in preorder (`allDepths`)

Vogliamo progettato 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`.
Proviamo ad adattare la visita in preorder, definendo la visita di un nodo v in questo modo: se v è radice si imposta la profondità a zero, altrimenti si imposta la profondità a $1+$ la profondità del padre, che è già impostata, dato che il padre è già stato visitato.

Algoritmo `allDepths(T,v)`
Input: $v \in T$, e `u.depth` impostato correttamente per u padre di v.
Output: `z.depth` impostato correttamente $\forall z \in T_v$.

```
//visita
if(T.isRoot(v)) then{
```

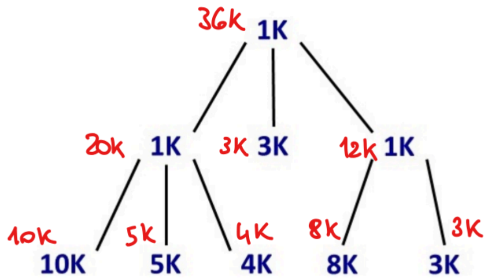
```
        v.depth <- 0;
    }
    else{
        v.depth(v) <- 1 + T.parent(v).depth;
    }

    //visita ricorsiva dei sottoalberi radicati nei figli
    forall w ∈ T.children(v) do{
        allDepths(w);
    }
```

> **Osservazioni**
>
> Per impostare il campo `depth` per tutti i nodi di T invoco `allDepths(T, T.root())`.
> `allDepths` non restituisce alcun valore (non c'è un `return`), ma modifica i modi dell'albero che si considerano come oggetti globali che sopravvivono all'esecuzione dell'algoritmo.

4.1.5.3.2. Esempi di algoritmi basati sulla visita in postoder

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.

Si consideri un file system gerarchico in cui la struttura è rappresentata da un albero T dove i nodi interni corrispondono alla 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`.



Nell'esempio sovrastante sono rappresentati in blu i valori dei campi `loc-size`, mentre in rosso quelli dei campi `aggr-size` alla fine dell'algoritmo.

Algoritmo `diskSpace(T,v)`
Input: $v \in T$ e `u.loc-size` impostato $\forall u \in T$.
Output: `v.aggr-size`, `z.aggr-size` impostato correttamente $\forall z \in T_v$.

```
v.aggr-size <- v.loc-size;
foraeach w ∈ T.children(v) do{
     v.aggr-size <. v-aggr-size + diskSpace(T,w);
}
return v.aggr-size;
```

Alternativamente si può scegliere di non far restituire `v.aggr-size` (con input e output analoghi):

```
v.aggr-size <- v.loc-size;
foraeach w ∈ T.children(v) do{
     diskSize(T,w);
     v.aggr-size <. v-aggr-size + w.aggr-size;
}
```

In questa versione la chiamata ricorsiva è isolata, infatti non restituisce nessun valore.

Osservazioni per la scrittura di algoritmi ricorsivi

- Un algoritmo ricorsivo può utilizzare sia *variabili globali* che sopravvivono a tutta l'esecuzione dell'algoritmo, sia *variabili locali alle singole invocazioni* che rimangono in vita solo durante l'esecuzione dell'invocazione corrispondente.
- Gli eventuali valori restituiti dalle invocazioni ricorsive (se ce ne sono) devono essere "utilizzati" in qualche modo, altrimenti vanno perduti.

4.1.5.5. Complessità

4.1.5.5.1. Complessità di `preorder(T.root())`

Sia n il numero di nodi di T. Consideriamo l'albero della ricorsione associato all'esecuzione di `preorder(T.root())`:

- Ha n nodi associati alle invocazioni ricorsive che sono *esattamente* una per ogni nodo di T;

- Il costo del nodo associato all'invocazione di `preorder(v)`, con $v \in T$ generico, è $\Theta(t_v + c_v + 1)$, dove t_v è il costo di "visita v" e c_v è il numero di figli di v.
  $\implies$ La complessità di `preorder(T.root())` è $\Theta\left(\sum_{v \in T}(t_v + c_v + 1)\right) = \Theta\left(\left(\sum_{v \in T} t_v\right) + \left(\sum_{v \in T}(c_v + 1)\right)\right) = \Theta\left(n + \sum_{v \in T} t_v\right)$.

> **Osservazione**
>
> Si ha la stessa complessità per la visita in `postorder` e `inorder` (per gli alberi binari).
> Dall'analisi discende che le visite consentono di enumerare tutti i nodi di T in tempo lineare in $|T|$ e, se $t_v \in \Theta(1)$, la complessità totale sarebbe $\Theta(n = |T|)$.

4.1.5.5.2. Complessità di `allDepths(T, T.root())`

`allDepths(T, T.root())` ha lo stesso schema della visita in `preorder` e $t_v \in \Theta(1) \forall v \in T$, quindi la complessità totale è $\Theta(n)$, dove $n = |T|$.

4.1.5.5.3. Complessità di `diskSum(T, T.root())`

`diskSum(T, T.root())` ha lo stesso schema della visita in `postorder` e $t_v \in \Theta(c_v + 1)$, con c_v il numero di figli di v. Allora la complessità è $T\left(n + \sum_{v \in T} t_v\right) = \Theta\left(n + \sum_{v \in T}(c_{v+1})\right) = \Theta(n)$.

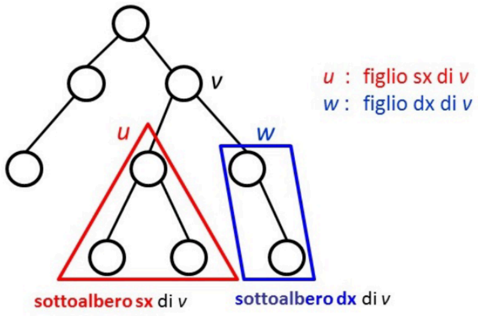
> **Riepilogo sugli alberi generali**
>
> - Definizione: albero, antenati, discendenti, sottoalbero, profondità di un nodo, altezza di un nodo, profondità di un nodo e di un albero;
> - Relazione tra altezza di un albero e profondità delle foglie;
> - Algoritmi per il calcolo della profondità e altezza di un nodo e dell'altezza di un albero;
> - Visite: preorder, postorder e le loro applicazioni;
> - Algoritmi di visita come template generali.

4.2. Alberi binari

> Per *arietà* di un albero si intende il *massimo numero di figli di un nodo interno*.

> Un *albero binario* T è un albero ordinato in cui:

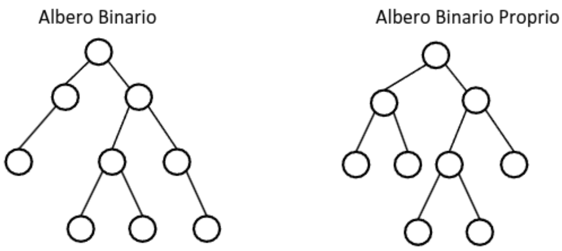
- Ogni *nodo interno ha* ≤ 2 *figli*;
- Ogni nodo non radice è etichettato come *figli sinistro* (sx) o *destro* (dx) di suo padre;
- Se ci sono entrambi i figli, il figlio sinistro viene prima del figlio destro nell'ordinamento dei figli di un nodo.



> Un *albero binario proprio* T è un albero binario tale che *ogni nodo interno ha* ***esattamente*** *2 figli*.

Terminologia

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



4.2.1. 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);
}
```

4.2.2. Proprietà

Sia T un albero binario proprio non vuota, dove $n = |T|$ è il numero di nodi in T, m (o n_E) è il numero di foglie in T, $n - m$ (o n_I) è il numero di nodi interni in T e h è l'altezza di T.

1. $m = n - m + 1$, ovvero le foglie sono uguali al numero di nodi interni più uno; ci permette di stabilire quanto spazio - al massimo - verrà occupato;
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. $\log_2(n + 1) - 1 \leq h \leq \frac{n-1}{2}$.