5 Alberi

5.0.1 Definizioni

Definizione 5.1 (Albero radicato – *rooted tree*)**.** *Un albero consiste di un insieme di nodi e un insieme di archi orientati che connettono coppie di nodi, con le seguenti proprietà:*

- *un nodo dell'albero è designato come nodo radice;*
- *ogni nodo n, a parte la radice, ha esattamente un arco entrante;*
- *esiste un cammino unico dalla radice ad ogni nodo;*
- *l'albero è connesso.*

Definizione 5.2 (Albero radicato, definizione ricorsiva)**.** *Un albero è dato da:*

- *un insieme vuoto, oppure*
- *una radice e zero o più sottoalberi, ognuno dei quali è albero; la radice è connessa alla radice di ogni sottoalbero con un arco orientato.*

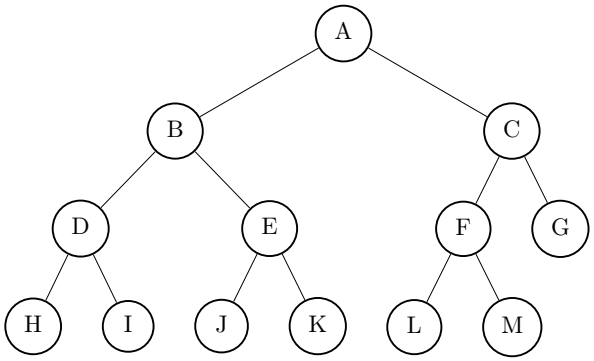
Definizione 5.3 (Profondità – *depth*)**.** *La lunghezza del cammino semplice dalla radice al nodo (misurato in archi).*

Definizione 5.4 (Livello – *level*)**.** *L'insieme dei nodi alla stessa profondità.*

Definizione 5.5 (Altezza dell'albero – *height*)**.** *La profondità massima delle sue foglie.*

5.1 Terminologia

- A è la radice (*root*);
- B, C sono radici dei sottoalberi (*roots of their subtrees*);
- D, E sono fratelli (*siblings*);
- D, E sono figli (*children*) di B;
- B è il padre (*parent*) di D, E;
- H, I, J, K, L, M, G sono foglie (*leafs*);
- gli altri nodi sono nodi interni (*internal nodes*);
- E è lo zio di I;
- B è il nonno di I, I è il nipote di B.



5.2 Alberi binari

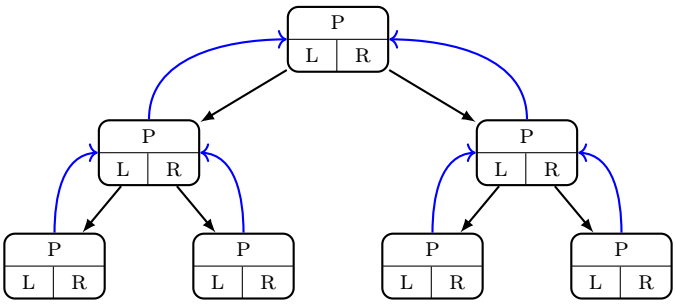
Definizione 5.6 (Albero binario)**.** *Un albero binario è un albero radicato in cui ogni nodo ha al massimo due figli, che vengono identificati come figlio sinistro e figlio destro.*

Nota. *Due alberi T e U che hanno gli stessi nodi, gli stessi figli per ogni nodo e la stessa radice, sono distinti qualora un nodo u sia designato come figlio sinistro di un nodo v in T come figlio destro del medesimo nodo in U. In altre parole, anche se due alberi hanno lo stesso numero di nodi ed ognuno di questi nodi ha lo stesso numero di figli non è che detto che l'albero risultante sia identico.*

Specifica BINARY TREE

```
// GESTIONE ALBERO

Tree(ITEM v)  // costruisce un nuovo nodo, contenente v, senza figli o genitori
ITEM read  // legge il valore memorizzato nel nodo
write(ITEM v)  // modifica il valore memorizzato nel nodo
TREE parent  // restituisce il padre, oppure nil se questo nodo è radice

// GESTIONE STRUTTURA

// restituiscono il figlio sinistro (destro) di questo nodo,
// restituisce nil se assente
TREE left
TREE right

// inserisce il sottoalbero radicato in t
// come figlio sinistro (destro) di questo nodo
insertLeft(TREE t)
insertRight(TREE t)

// distrugge (ricorsivamente) il figlio sinistro (destro) di questo nodo
deleteLeft
deleteRight
```

5.2.1 Memorizzazione di un albero binario



Vengono memorizzati i seguenti campi:

- *parent*: riferimento al nodo padre;
- *left*: riferimento al figlio sinistro;
- *right*: riferimento al figlio destro.

Uno qualunque di questi oggetti potrebbe essere pari a **nil**, stando ad indicare che sotto di sé non esiste nessun sottoalbero.

5.2.2 Implementazione

Implementazione BINARY TREE in pseudocodice

```
TREE Tree(ITEM v)
    TREE t = new TREE
    t.parent ← nil
    t.left ← t.right ← nil
    t.value ← v
    ritorna t
insertLeft(TREE t)
    se left ≠ nil allora
        t.parent ← this
        left ← t
insertRight(TREE t)
    se right ≠ nil allora
        t.parent ← this
        right ← t
```

```
deleteLeft(TREE t)
    se left ≠ nil allora
        left.deleteLeft
        left.deleteRight
        left ← nil
deleteRight(TREE t)
    se right ≠ nil allora
        right.deleteLeft
        right.deleteRight
        right ← nil
```

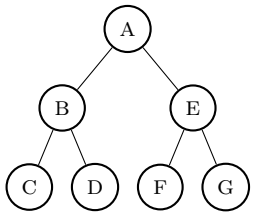
5.2.3 Visite

La visita di un albero (o la ricerca) è una strategia per passare attraverso (visitare) tutti i nodi di un albero. Si possono distinguere due tipi di visite:

1. visita in profondità: chiamata anche *Deep-First Search* (DFS), per visitare un albero visita ricorsivamente ognuno dei suoi sottoalberi; esistono tre varianti in base a quando il nodo viene visitato (in pre, in o post order); questa particolare visita richiede il meccanismo di una pila (*stack*);
2. visita in ampiezza: chiamata anche *Breadth First Search* (BFS), per visitare un albero visita ogni livello, uno dopo l'altro partendo dalla radice; richiede il meccanismo di una coda (*queue*).

A seconda di dove scrivo il codice in questo schema ottengo una visita diversa.

```
dfs-schema(TREE t)
    se t ≠ nil allora
        // pre-order visit
        stampa t
        dfs(t.left)
        // in-order visit
        stampa t
        dfs(t.right)
        // post-order visit
        stampa t
```



pre-visita	ABCDEFG
in-visita	CBDAFEG
post-visita	CDBFGEA

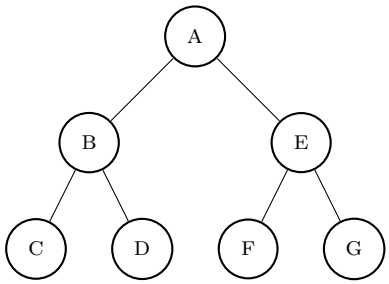
5.2.4 Applicazioni

In genere post-visita e in-visita sono quelle più applicate, la pre-visita meno.

Visita in post-ordine

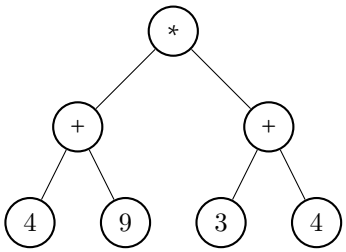
Una possibile applicazione della visita post-ordine è quella di effettuare un conteggio dei nodi presenti nell'albero.

```
count(TREE t)
    se t == nil allora
        // è un albero vuoto
        ritorna 0
    allora
        // conto ricorsivamente i nodi
        C_ℓ = count(t.left)
        C_r = count(t.right)
        ritorna C_ℓ + C_r + 1
```



Visita in ordine (in-visita)

```
int stampaEspressioni(TREE t)
    se t.left == nil and t.right == nil allora
        // siamo in una foglia
        stampa t.read
    allora
        // sono su un nodo interno
        stampa "("
        stampaEspressioni(t.left)
        stampa t.read
        stampaEspressioni(t.right)
        stampa ")"
```



Stampa: `( 4 + 9 ) * ( 3 + 4 )`

Complessità di una visita

Il costo di una visita di un albero contenente n nodi è $\Theta(n)$, in quanto ogni nodo viene visitato al massimo una volta.

5.3 Alberi generici

Specifica GENERIC TREE

```
// GESTIONE ALBERO
Tree(ITEM v)  // costruisce un nuovo nodo, contenente v, senza figli o genitori
ITEM read  // legge il valore memorizzato nel nodo
write(ITEM v)  // modifica il valore memorizzato nel nodo
TREE parent  // restituisce il padre, oppure nil se questo nodo è radice

// GESTIONE STRUTTURA
// restituiscono il primo figlio,                 // inserisce il sottoalbero t
// oppure nil se questo nodo è una foglia         // come primo figlio di questo nodo
TREE leftmostChild                                insertSibling(TREE t)

// restituisce il prossimo fratello,              // distuggi l'albero radicato
// oppure nil se assente                          // identificato dal primo figlio
TREE rightSibling                                 deleteChild

// inserisce il sottoalbero t                     // distuggi l'albero radicato
// come primo nodo di questo nodo                 // identificato dal primo figlio
insertChild(TREE t)                               deleteSibling
```

5.3.1 Visita in profondità

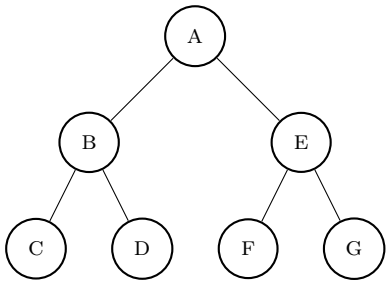
Un albero binario è anche un albero generale e lo visitiamo esattamente come lo visitavamo prima.

```
dfs(TREE t)
    se t ≠ nil allora
        // pre-order visit
        stampa t
        dfs(t.left)
        // effettuo visita
        TREE u ← t.leftmostChild
        finché u ≠ nil fai
            dfs(u)
            u.rightSibling
        // post-order visit
        stampa t
```

5.3.2 Visita in ampiezza

Mentre nella visita in profondità la pila (*stack*) era implicita nella chiamata ricorsiva, in questo caso è necessario utilizzare *esplicitamente* una coda (*queue*). Un'altra differenza fra i due algoritmi è che l'algoritmo di visita in profondità era un algoritmo ricorsivo, mentre questo è iterativo.

```
bfs(Tree t)
    Queue Q ← Queue
    Q.enqueue(t)
    // inserisci la radice
    finché not Q.isEmpty fai
        // fintanto che la coda non è vuota
        // estraggo un nodo dalla coda
        Tree u ← Q.dequeue
        // visita per livelli del nodo u
        stampa u
        // fintanto che ho almeno un figlio
        u ← u.leftmostChild
        finché u ≠ nil fai
            // metto in coda il figlio
            Q.dequeue(u)
            // passo al figlio destro
            u ← u.leftmostChild
```



Sequenza: A B E C D F G

Commento Mettiamo in coda tutti i nodi che vogliamo visitare passo passo. Qui la stampa è in pre-visita ma qui – a differenza dei grafi – non ha molta importanza se la visita la facciamo prima o dopo. Visito tutti i figli prima di passare al livello successivo.

5.4 Memorizzazione

Esistono diversi modi per memorizzare un albero, più o meno indicati a seconda del numero massimo e medio di figli presenti. Le realizzazioni possibili sono:

1. con vettore dei figli;
2. primo figlio, prossimo fratello;
3. con vettore dei padri

5.4.1 Realizzazione con vettore dei figli

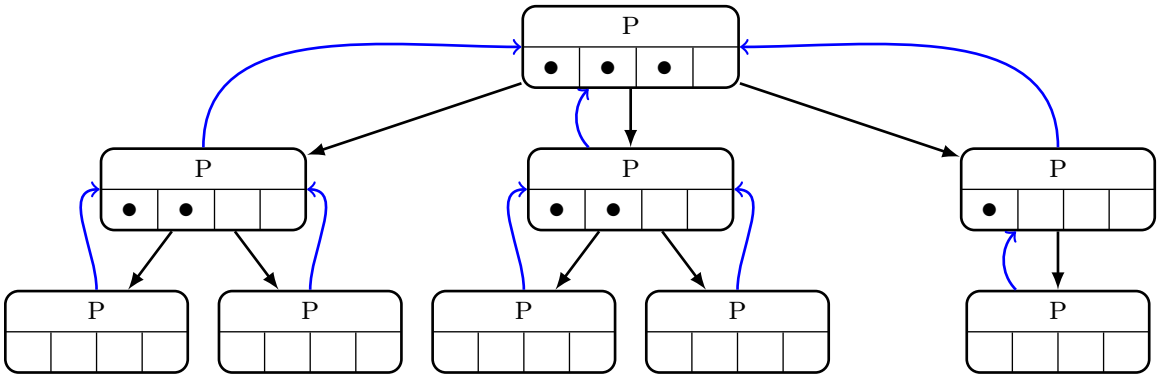


Figure 1: Realizzazione con vettore dei figli

Vengono memorizzati i seguenti campi:

- *parent* che è il riferimento al nodo padre;
- vettore dei figli il quale a seconda del numero dei figli può comportare una discreta quantità di spazio sprecato.

5.4.2 Realizzazione basata su Primo figlio, prossimo fratello

Viene implementato come una lista di fratelli.

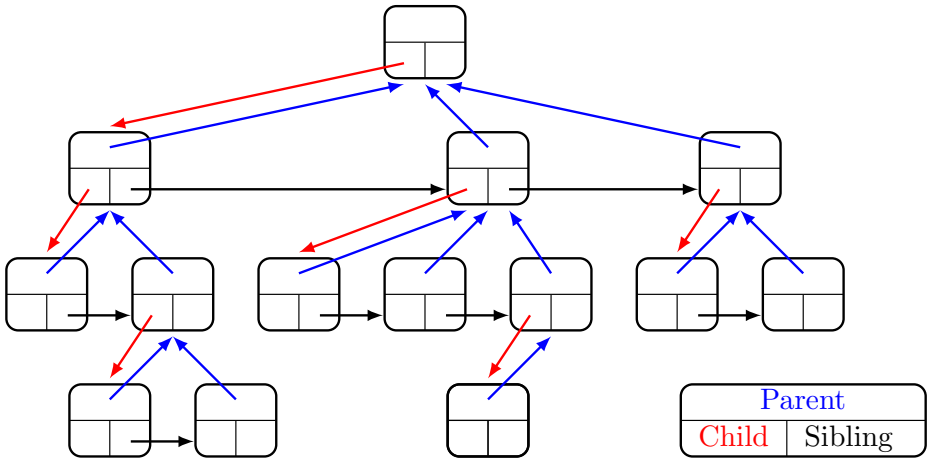


Figure 2: Realizzazione basata su Primo figlio, prossimo fratello

La memorizzazione che viene utilizzata nel file system è esattamente quella che ci siamo dati.

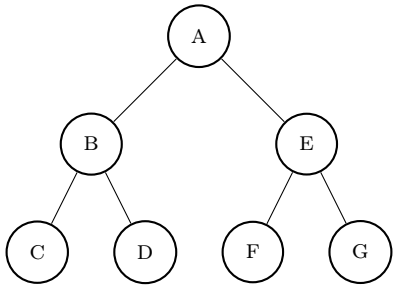
Implementazione TREE "primo figlio, prossimo fratello" in pseudocodice

```
TREE parent            // Riferimento al padre
TREE child             // Riferimento al primo figlio
TREE sibling           // Riferimento al prossimo fratello
ITEM value             // Valore memorizzato nel nodo

TREE Tree(ITEM v)
    TREE t = new TREE
    t.value ← v
    t.parent ← t.child ← t.sibling ← nil
    ritorna t

insertChild(TREE t)
    t.parent ← self
    // inserisci t prima dell'attuale primo
       figlio
    t.sibling ← child
    child ← t

insertChild(TREE t)
    t.parent ← parent
    // inserisci t prima dell'attuale prossimo
       fratello
    t.sibling ← sibling
    sibling ← t

deleteChild
    TREE newChild ← child.rightSibling
    delete(child)
    child ← newChild

deleteSibling
    TREE newBrother ← sibling.rightSibling
    delete(sibling)
    sibling ← newBrother

delete(TREE t)
    TREE u ← t.leftmostChild
    finché u ≠ nil fai
        TREE next ← u.rightSibling
        delete(u)
        u ← next
```

5.4.3 Realizzazione con vettore dei padri

Nella Realizzazione con vettore dei padri l'albero è rappresentato da un vettore i cui elementi contengono il valore associato al nodo e l'indice della posizione del padre del vettore.

1	a	0
2	B	1
3	E	1
4	C	2
5	D	2
6	F	3
7	G	3



Questa realizzazione può sembrare particolarmente assurda poiché dato un nodo non permette di stabilire direttamente quali sono i suoi figli, ma ci sono molti algoritmi che sono interessati solo ai padri. Questa la rappresentazione più compatta che possiamo creare, vedremo la sua utilità quando andremo a studiare le visite sui grafi.