

## 5 Alberi

### 5.1 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.

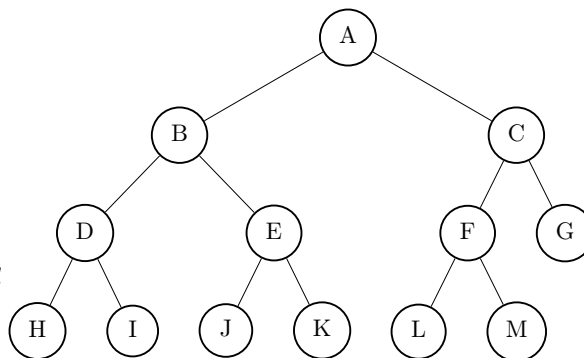
**Definizione 5.3** (Profondità – *depth*). La lunghezza del cammino semplice dalla radice al nodo (misurato in archi).

**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.

### 5.2 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 (*leaves*);
- gli altri nodi sono nodi interni (*internal nodes*);
- $E$  è lo zio (il fratello del padre) di  $I$ ;
- $B$  è il nonno di  $I$ ,  $I$  è il nipote di  $B$ .



### 5.3 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.

```
// 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

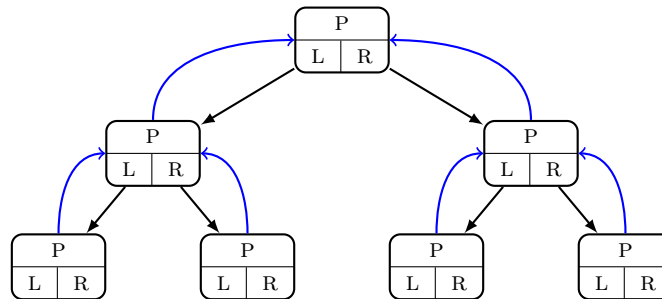
// GESTIONE STRUTTURA
// restituiscono il figlio sinistro (destro) di questo nodo,
// restituisce nil se assente
TREE left
TREE right

// 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
```

*Nota.* Le funzioni *senza parametri* sono indicate con un carattere senza grazie e privi di parentesi tonde vuote al fine di alleggerire la lettura del codice.

### 5.3.1 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 non esiste nessun sottoalbero.

### 5.3.2 Implementazione

**Algoritmo 5.1:** Implementazione albero binario in pseudocodice

---

```
// crea un nuovo albero
// restituisce la radice dell'albero creato
TREE Tree(ITEM v)
|   TREE t = new TREE
|   t.parent ← nil
|   t.left ← t.right ← nil
|   t.value ← v
|   return t
insertLeft(TREE t)
|   if left ≠ nil then
|       |   t.parent ← this
|       |   left ← t
insertRight(TREE t)
|   if right ≠ nil then
|       |   t.parent ← this
|       |   right ← t
// elimina ricorsivamente il sottoalbero sinistro
deleteLeft
|   if left ≠ nil then
|       |   left.deleteLeft
|       |   left.deleteRight
|       |   left ← nil
// elimina ricorsivamente il sottoalbero destro
deleteRight
|   if right ≠ nil then
|       |   right.deleteLeft
|       |   right.deleteRight
|       |   right ← nil
```

---

### 5.3.3 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:

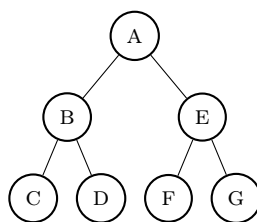
1. visita in profondità: chiamata anche *Depth-First Search* (DFS), per visitare un albero visita ricorsivamente ognuno dei suoi sottoalberi; esistono tre varianti in base a quando il nodo viene visitato (pre, in o post-ordine); questa particolare visita sfrutta implicitamente il meccanismo di una pila (*stack*) tramite le chiamate ricorsive effettuate;
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 esplicitamente l'utilizzo di una coda (*queue*).

**Algoritmo 5.2:** Schema per visita in profondità

---

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

---



pre-visita	A B C D E F G
in-visita	C B D A F E G
post-visita	C D B F G E A

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

### 5.3.4 Applicazioni

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

#### Visita in post-ordine

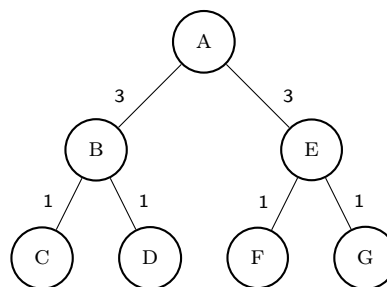
Una possibile applicazione della visita post-ordine è quella di effettuare un conteggio dei nodi presenti nell'albero.

---

**Algoritmo 5.3:** Conteggio dei nodi in un albero

---

```
count(TREE t)
|   if t == nil then
|       // è un albero vuoto
|       return 0
|   else
|       // conto ricorsivamente i nodi
|       Cℓ = count(t.left)
|       Cr = count(t.right)
|       return Cℓ + Cr + 1
```



#### Visita in ordine (in-visita)

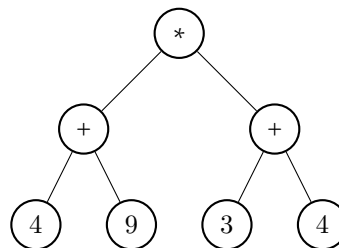
Una possibile applicazione della visita post-ordine è quella di stampare espressioni con operatori binari.

---

**Algoritmo 5.4:** Stampa espressioni con operatori binari

---

```
int stampaEspressioni(TREE t)
|   if t.left == nil and t.right == nil then
|       // siamo in una foglia
|       stampa t.read
|   else
|       // 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.

## 5.4 Alberi generici

---

**Algoritmo 5.5:** Specifica albero generico

---

```
// 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 prossimo fratello di questo nodo
TREE leftmostChild insertSibling(TREE t)

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

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

---

### 5.4.1 Visita in profondità

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

---

**Algoritmo 5.6:** Schema per visita in profondità

---

```
dfs(TREE t)
|
|   if t ≠ nil then
|       // pre-order visit
|       stampa t
|       dfs(t.left())
|       // effettuo visita
|       TREE u ← t.leftmostChild
|       while u ≠ nil do
|           dfs(u)
|           u.rightSibling
|       // post-order visit
|       stampa t
|
```

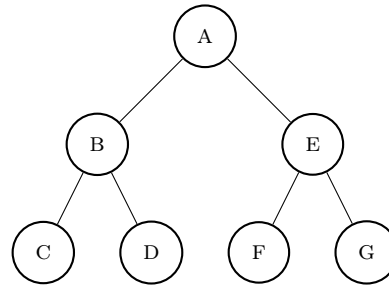
---

### 5.4.2 Visita in ampiezza

Mentre nella visita in profondità il meccanismo della pila (*stack*) era implicito nelle chiamate ricorsive, in questo caso è necessario utilizzare *esplicitamente* una coda (*queue*). Un'altra differenza fra i due algoritmi è che quello in profondità è un algoritmo ricorsivo, l'altro è iterativo. Quando tutti i nodi di un livello vengono estratti dalla coda, la coda contiene solo ed unicamente i nodi del livello successivo.

```

bfs(TREE t)
  QUEUE Q ← Queue
  Q.enqueue(t) // inserisci la radice
  while not Q.isEmpty do
    // 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
    while u ≠ nil do
      // metto in coda il figlio
      Q.enqueue(u)
      // passo al figlio destro
      u ← u.rightSibling
  
```



Sequenza: A B E C D F G

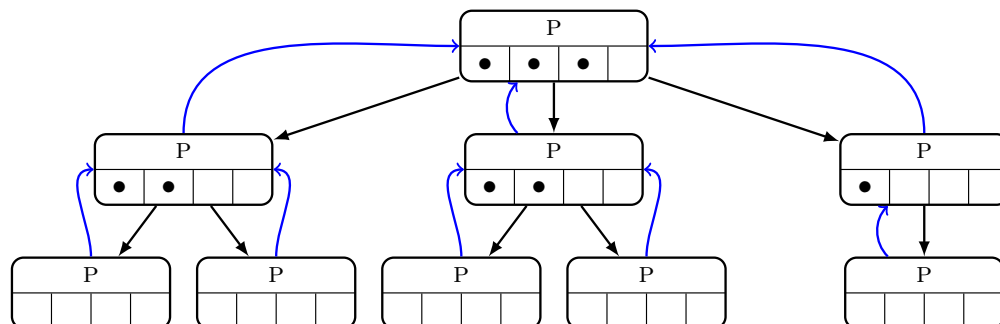
**Commento** Mettiamo in coda tutti i nodi che vogliamo visitare passo passo. Qui la stampa è in pre-visita ma qui – a differenza dei grafi – non ha molta importanza se la visita la facciamo prima o dopo. Visito tutti i figli prima di passare al livello successivo.

## 5.5 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

### 5.5.1 Realizzazione con vettore dei figli



**Figure 1:** Realizzazione con vettore dei figli

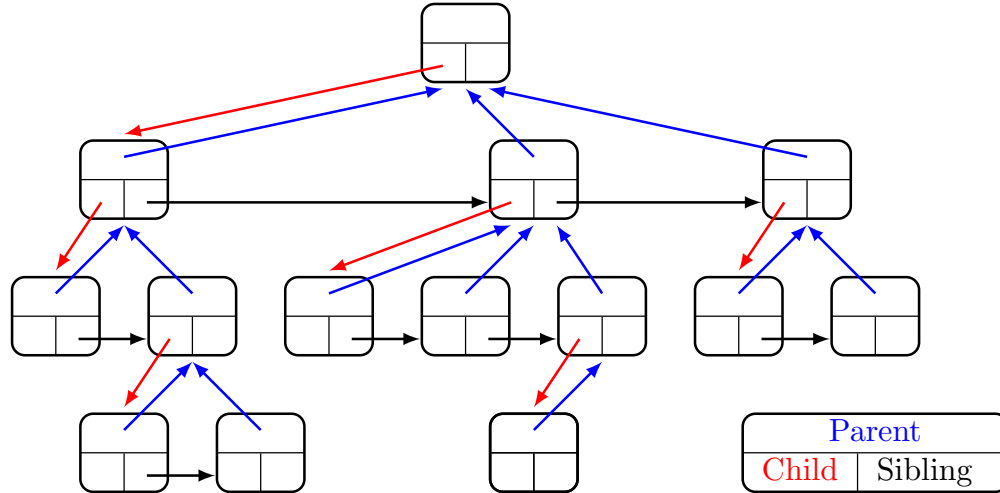
Vengono memorizzati i seguenti campi:

- *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.

### 5.5.2 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 questa.

---

#### Algoritmo 5.8: Implementazione albero “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
    return t

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

insertSibling(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

// metodo ausiliare
delete(TREE t)
    TREE u ← t.leftmostChild
    while u ≠ nil do
        TREE next ← u.rightSibling
        delete(u)
        u ← next

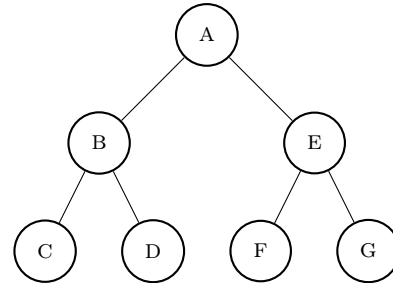
```

---

### 5.5.3 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 è la rappresentazione più compatta che possiamo creare, vedremo la sua utilità quando andremo a studiare le visite sui grafi.