Alberi

April 16, 2020

Obiettivi: albero come struttura ricorsiva, sviluppo di algoritmi ricorsivi su alberi.
Argomenti: definizione, terminologia e rappresentazione di alberi, calcolo di altezza e cardinalità, visite.

1 Definizione, terminologia e rappresentazione

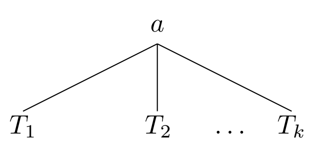
Sia A un insieme (l'insieme delle etichette). L'insieme di alberi su A, denotato con $T(A)$, è definito induttivamente come segue:

$$a \in A \wedge T_1 \in T(A) \wedge T_2 \in T(A) \wedge \cdots \wedge T_k \in T(A) \quad \text{con } k \geq 0$$

$$\Downarrow$$

$$\{a, T_1, T_2, ..., T_k\} \in T(A)$$

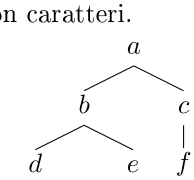
di cui interpretazione è la seguente: dato un nodo con etichetta $a \in A$ è k alberi, si può formare un albero agganciando i k alberi al nodo. (Nella definizione precedente $\{a, T_1, T_2, ..., T_k\}$ denota un albero e non un insieme.) Graficamente:



Dalla definizione, con $k = 0$, segue che un singolo nodo con etichetta in A è un albero. Gli altri alberi in $T(A)$ si possono costruire a partire dagli alberi che contengono un nodo solo. Seconda la definizione precedente l'albero vuoto (albero con zero nodi) non fa parte di $T(A)$. La definizione permette invece di avere alberi in cui più nodi hanno la stessa etichetta. In certi casi conviene includere l'albero vuoto nell'insieme e/o escludere la possibilità di aver più nodi con la stessa etichetta.

Un albero è un **grafo connesso aciclico**. (Connesso vuole dire che esiste un cammino fra qualunque coppia di nodi.) Una **foresta** è un insieme di alberi.

Consideriamo il seguente albero etichettato con caratteri.



Il nodo a è la **radice** dell'albero. I nodi d, e e f sono **foglie**. I nodi b e c sono **nodi interni**. Il nodo a è **padre** di b e c. Il nodo b è **figlio** di a. Il nodo e è **fratello** del nodo d. Il nodo f è **discendente** del nodo a. Il nodo a è **avo** di f.

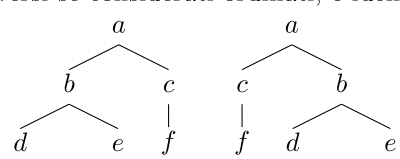
Un cammino dalla radice ad una foglia è un **ramo**. Il **livello** di un nodo è il numero degli archi del cammino che porta al nodo dalla radice (livello della radice è 0). L'**altezza** di un albero è il livello del nodo che ha livello massimo fra i nodi (per l'esempio precedente è 2). Il **grado** è il numero di figli del nodo che ha il più grande numero di figli (il grado è 2 per l'esempio precedente).

In certi applicazione la radice può essere visto come una **sorgente** (dalla quale viene distribuita qualcosa verso le foglie) oppure come un **pozzo** (che raccoglie qualcosa che arriva dalle foglie).

Gli alberi si possono descrivere con una stringa. L'albero precedente corrisponde a

$$\{a, \{b, \{d\}, \{e\}\}, \{c, \{f\}\}\}$$

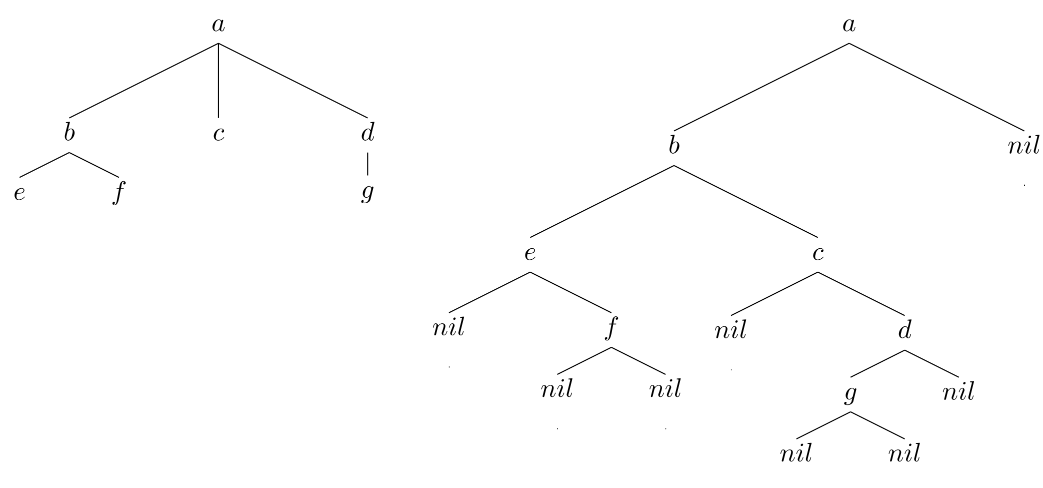
L'albero viene detto **ordinato** se l'ordine in cui appaiono i figli di un nodo conta. I due alberi sotto sono diversi se considerati ordinati, e identici se considerati non ordinati.



Rappresentazione "naturale" di un albero di grado k: ogni nodo contiene l'etichetta e k puntatori che fanno riferimenti ai figli (se un nodo ha meno di k figli allora una parte dei puntatori è nil).

Rappresentazione binaria di un albero di grado k: ogni nodo contiene l'etichetta, un primo puntatore al **primo figlio** (detto **child**) e un secondo puntatore al **fratello successivo** (detto **sibling**). (Naturalmente, anche in questo caso, child e/o sibling possono essere nil.)

Seguono due alberi. Sulla sinistra un albero di grado 3. Sulla destra la sua rappresentazione binaria (dove vengono indicati esplicitamente i puntatori nil).



Gli **alberi binari posizionali** sono alberi binari (cioè di grado 2) in cui conta l'ordine dei nodi. Definizione induttiva dell'insieme, $BT(A)$, degli alberi binari posizionali su un insieme A:

a) $\emptyset \in BT(A)$ (l'albero vuoto fa parte dell'insieme)

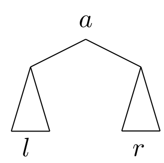
b)

$$a \in A \wedge l \in TB(A) \wedge r \in TB(A)$$

$$\Downarrow$$

$$\{a, l, r\} \in TB(A)$$

Cioè, data un'etichetta, a, e due alberi, l e r in $TB(A)$, agganciando l come sottoalbero sinistro e r come sottoalbero destro al nodo con etichetta a, si ottiene un nuovo albero. (In questo caso la definizione include esplicitamente l'albero vuoto nell'insieme $TB(A)$.) Graficamente:



Rappresentazione di un albero binario posizionale: ogni nodo contiene l'etichetta e due puntatori, $left$ e $right$, che fanno riferimento al sottoalbero sinistro e destro.

2 Algoritmi di base

La **cardinalità** di un albero è il numero dei suoi nodi.

Algoritmo per determinare la cardinalità di un albero binario posizionale:

```
2TREE-CARD(T)
if T = nil then
    return 0
else
    l ← 2TREE-CARD(T.left)
    r ← 2TREE-CARD(T.right)
    return l + r + 1
end if
```

Algoritmo per determinare la cardinalità di un albero rappresentato con *child* e *sibling*:

```
kTREE-CARD(T)
if T = nil then
    return 0
else
    card ← 1
    C ← T.child
    while C ≠ nil do
        card ← card+ kTREE-CARD(C)
        C ← C.sibling
    end while
    return card
end if
```

Algoritmo per determinare l'altezza di un albero binario posizionale:

```
2TREE-HIGHT(T)        ▷ pre: T non è vuoto
if T.left = nil and T.right = nil then
    return 0        ▷ T ha un solo nodo
else
    hl, hr ← 0
    if T.left ≠ nil then
        hl ← 2TREE-HIGHT(T.left)
    end if
    if T.right ≠ nil then
        hr ← 2TREE-HIGHT(T.right)
    end if
    return 1 + max{hl, hr}
end if
```

Algoritmo per determinare la cardinalità di un albero rappresentato con *child* e *sibling*:

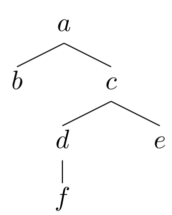
```
kTree-Hight(T)      ▷ pre: T non è vuoto
if T.child = nil then
    return 0      ▷ T ha un solo nodo
else
    h ← 0
    C ← T.child
    while C ≠ nil do
        h ← max{h, kTree-Hight(C)}
        C ← C.sibling
    end while
    return h + 1
end if
```

2.1 Visite

La visita (completa) di un albero consiste in un'ispezione dei nodi dell'albero in cui ciascun nodo sia "visitato" (ispezionato) esattamente una volta. Due tipi di visite:

- **Visita in profondità** (Depth First Search, DFS): lungo i rami, dalla radice alle foglie.
- **Visita in ampiezza** (Breadth First Search, BFS): per livelli, da quello della radice in poi.

Consideriamo il seguente grafo.



Visite in profondità del grafo precedente (preordine destro): a, c, e, d, f, b.
Visite in profondità del grafo precedente (preordine sinistra): a, b, c, d, f, e.
Visite in ampiezza del grafo precedente (livelli da sinistra a destra): a, b, c, d, e, f.
Visite in ampiezza del grafo precedente (livelli da sinistra a destra): a, c, b, e, d, f.

La **visita in profondità** scende lungo un ramo fino ad una foglia, poi, tornando su, ricomincia a scendere appena ci sono dei nodi ancora non visitati. Scendendo lungo un ramo, si possono quindi memorizzare in una **pila** i nodi da dove ricominciare la discesa. Il seguente algoritmo effettua la visita in profondità su un grafo in cui ogni nodo contiene riferimenti diretti ad ogni figlio (rappresentazione "naturale").

```
Tree-DFS-Stack(T)      ▷ pre: T non è vuoto
S ← pila vuota
Push(S, T)
while ¬ Empty(S) do
    T' ← Pop(S)
    visita T'
    for all C figlio di T' do
        Push(S, C)
    end for
end while
```

Simulazione dell'algoritmo sul grafo precedente: la seguente tabella riporta il contenuto della pila dopo ogni operazione (Push o Pop) sulla pila. Gli elementi con asterisco sono quelli che stanno per esseri tolti con un Pop e quindi visitati consecutivamente. Gli elementi entrano nella pila da sopra e escono verso su.

						e^*					
			c^*		d	d	d^*		f^*		
a^*		b	b	b	b	b	b	b	b	b^*	

La **visita in ampiezza** deve visitare l'albero livello per livello. Questo si può ottenere con una **coda**. Il seguente algoritmo effettua la visita in ampiezza su un grafo in cui ogni nodo contiene riferimenti diretti ad ogni figlio (rappresentazione "naturale").

```
Tree-BFS-Queue(T)          ▷ pre: T non è vuoto
Q ← code vuota
Enqueue(Q, T)
while ¬ Empty(Q) do
    T′ ← Dequeue(Q)
    visita T′
    for all C figlio di T′ do
        Enqueue(Q, C)
    end for
end while
```

Simulazione dell'algoritmo sul grafo precedente: la seguente tabella riporta il contenuto della coda dopo ogni operazione (Enqueue o Dequeue) sulla coda. Gli elementi con asterisco sono quelli che stanno per esseri tolti con un Dequeue e quindi visitati consecutivamente. Gli elementi entrano nella coda da sinistra e escono verso destra.

$$
\begin{array}{cccc}
 & \to & a^* & \to \\
 & \to & & \to \\
 & \to & b & \to \\
\to & c & b^* & \to \\
 & \to & c^* & \to \\
 & \to & & \to \\
 & \to & d & \to \\
\to & e & d^* & \to \\
 & \to & e & \to \\
\to & f & e^* & \to \\
 & \to & f^* & \to \\
 & \to & & \to
\end{array}
$$

La visita in profondità può essere effettuata in maniera ricorsiva. Il seguente algoritmo effettua la visita in profondità su un grafo rappresentato con puntatori *child* e *sibling*.

```
Tree-DFS(T)          ▷ pre: T non è vuoto
visita T.key
C ← T.child
while C ≠ nil do
    Tree-DFS(C)
    C ← C.sibling
end while
```

(Provare a simulare su qualche grafo non banale.)

Complessità delle varie visite:

- per trovare un limite superiore possiamo contare quante operazioni Push/Pop (oppure Enqueue/Dequeue) avvengono in una DFS (o BFS);
- per ogni nodo si fa un inserimento ed una estrazione nella struttura dati d'appoggio (pila o coda);
- quindi DFS e BFS hanno costo $O(2n) = O(n)$ dove n è la cardinalità dell'albero.

Gli algoritmi precedenti che determinano la cardinalità e l'altezza di un albero si basano praticamente su una visita e quindi hanno anche loro complessità $O(n)$.