Alberi

Luca Tagliavini

March 11-15, 2021

Contents

1 Alberi

1.1 Intruduzione

Fin'ora abbiamo visto strutture dati sequenziali. Ad un elemento seguiva un'altro. Gli alberi ci consento di strutturare i dati in modo gerarchicho. Eccone la definizione formale:

- *Insieme vuoto di nodi*, oppure
- *Una radice* R e *zero o piu' alberi* disgiunti (detti *sottoalberi*), dove la radice R e' collegata alla radice di ogni sottoalbero.

I tipi di alberi piu' standard sono gli **alberi binari** che consentono 0, 1 o al piu' due (da cui deriva binario) sottoalberi. Ecco alcune terminologie:

- I nodi che non hanno nessun figlio sono chiamati **foglie**.
- L'albero soprastante al nodo considerato viene definito **genitore**.
- Negli alberi binari, avendo sempre al piu' due sottoalberi, possiamo riferici ad essi come **left node** e **right node**.
- **profondita'**: e' il numero di *archi* attraversati per andare dalla radice al nodo o viceversa. (dunque il nodo radice ha profondita' 0, i suo figli 1, etc...)
- **livello**: E' l'insieme di nodi alla stessa profondita'.
- **altezza**: Massima profondita' dell'albero.

1.2 Visita

Per le strutture sequenziali come le liste o gli array le funzioni di visita sono intuitive: si puo' partire dalla fine o dall'inizio e procedere nella direzione scelta. Per gli alberi abbiamo invece svariate tecniche per visitare tutti i nodi dell'albero:

- **depth-first search** (DFS, in profondita'): vengono visitati tutti i sottonodi uno dopo l'altro (si visita la prima foglia fino in fondo, poi la seconda fino in fondo, etc...). Si puo' svolgere in tre modi diversi:
  - **pre-ordine**: Visitiamo prima la radice (il nodo passato alla funzione) e svolgiamo una chiamata ricorsiva sul sotto albero sinistro e una su quello destro.
  - **in-ordine**: Facciamo la chiamata ricorsiva sul nodo di sinistra, poi visitiamo la radice (passato alla funzione) e poi quello di destra.
  - **post-ordine**: Simile alla pre-ordine ma il nodo radice viene lasciato per ultimo. Si fa dunque la chiamata ricorsiva sul nodo di sinitra, poi su quello di destra ed in fine si visita la radice.

- **breadth-first search** (BFS, in ampiezza): vengono visitati tutti i nodi sullo stesso livello, paretendo dall'insieme di nodi al livello 1, poi quelli al livello 2, etc.
  la BFS viene implementata tramite delle code.

  Gli algoritmi ricorsivi sono quelli piu' adatti per la visita di alberi (a meno di vincoli inposti da superiori).

1.2.1 Implementazione di DFS (pre)

```
visit(Tree t, Visitor v) -> void
  if t != NULL
    v(t)
    visit(t.left)
    visit(t.right)
```

1.2.2 Implementazione di DFS (in)

```
visit(Tree t, Visitor v) -> void
  if t!= NULL
    visit(t.left)
    v(t)
    visit(t.right)
```

1.2.3 Implementazione di DFS (post)

```
visit(Tree t, Visitor v) -> void
  if t!= NULL
    visit(t.left)
    visit(t.right)
    v(t)
```

1.2.4 Implementazione di BFS

```
visit(Tree t, Visitor v) -> void
  Queue q = new Queue()
  q.insert(t)
  while(not q.empty())
    Tree node = q.dequeue()
    v(node)

    if node.left != NULL
      q.insert(node.left)

    if node.right != NULL
      q.insert(node.right)
```

2 Alberi binari di ricerca

Idea: portare la ricerca binaria sugli array negli alberi binari.

Implementazione: ogni nodo v contiene due sottoalberi dove uno ha tutti i valori $\leq$ (per esempio quello di destra) e l'altro ramo i valori >= (per esempio quello di sinistra). La definizione va applicata ricorsivamente, dunque se avessimo un nodo con valore 10, e un sottoalbero con valore 6, che tuttavia ha come sottonodi valori maggiori di 10 come ad esempio 12 non sarebbe un albero valido per definizione. Insomma la regola del maggiore/minore deve essere verificata in tutti gli alberi e *sottoalberi* dei nodi di destra/sinistra.

2.1 Alberi AVL (Adelson-Velsky, Landis)

Un albero AVL e' una struttura alborescente *quasi* bilanciato. E' il primo approccio proposto il letteratura per alberi in grado di auto bilanciarsi. Fu Fu sviluppato da scienziati russi dell'USSR e in seguito tradotto per il resto del mondo. L'obbiettivo e' avere operazioni d'inserimento e rimozione autobilancianti con costo pessimo di $O(\log n)$. Ora capiremo come. Terminologia:

- **fattore di bilanciamento**: Indicato con $\beta(v)$ di un nodo v e' dato dalla differenza tra l'altezza del sottoalbero sinistro e del sottoalbero destro di v: $\beta(v) = altezza(v.left) - altezza(v.right)$.
- **Bilanciamento in altezza**: un albero si dice bilanciato in altezza se le altezze dei sottoalberi sinistro e destor in ogni nodo differiscono al piu' di uno. In altre parole, un albero e' bilanciato in altezza se $\forall v \in Nodi.|\beta(v)| \leq 1$.

Definizione
Un albero AVL e' un albero binario di ricerca bilanciato in altezza.

2.1.1 Siamo sicuri che abbia complessita' $O(n)$?

Proviamo a costuire *alberi di fibonacci*, ossia alberi che hanno in ogni nodo (eccetto le foglie) $\beta(v) = 1$, ossia ogni albero (di sinistra per dire) ha profondita' +1 rispetto a quello (di destra).
Questa e' la configurazione piu' sbilanciata che possiamo ottenere continuando ad aderire alle regole degli AVL.

Per costruzione si ha che il numero di nodi (n) dell'h-esimo sottoalbero ha valore:

$$n_h = n_{h-1} + n_{h-2} + 1$$

Dimostriamo che (F_n e' l'n-esimo numero di fibonacci $F_1 = F_2 = 1$):

$$n_h = F_{h+3} - 1$$

Dimostrazione per induzione:

- **Caso base** ($h = 0$): $n_0 = 1, F_3 = 2$
- **Caso base** ($h = 1$): $n_1 = 2, F_3 = 3$
- **passo induttivo**:

$$\begin{aligned} n_h &= n_{h-1} + n_{h-2} + 1 \\ &= (F_{h+2} - 1) + (F_{h+1} - 1) + 1 \\ &= F_{h+2} + F_{h+1} - 1 \\ &\text{per definizione di successione di Fibonacci} \\ &= F_{h+3} - 1 \end{aligned}$$

□

Attraverso dei conti (slide) si arriva a far vedere che la profonita' massima di un albero AVL sara' sempre, anche nel peggior caso qui' analizzato con la costruzione di Fibonacci $O(\log n)$.