Esercizi Capitolo 6 - Alberi binari di ricerca

Alberto Montresor

19 Agosto, 2014

Alcuni degli esercizi che seguono sono associati alle rispettive soluzioni. Se il vostro lettore PDF lo consente, è possibile saltare alle rispettive soluzioni tramite collegamenti ipertestuali. Altrimenti, fate riferimento ai titoli degli esercizi. Ovviamente, si consiglia di provare a risolvere gli esercizi personalmente, prima di guardare la soluzione.
Per molti di questi esercizi l'ispirazione è stata presa dal web. In alcuni casi non è possibile risalire alla fonte originale. Gli autori originali possono richiedere la rimozione di un esercizio o l'aggiunta di una nota di riconoscimento scrivendo ad `alberto.montresor@unitn.it`.

1 Problemi

1.1 Versione ricorsiva di insertNode() (Esercizio 6.2 del libro)

Si scriva una versione ricorsiva della procedura insertNode().

Soluzione: Sezione 2.1

1.2 Verifica ABR (Esercizio 6.3 del libro)

Si scriva una funzione che verifichi se un albero binario è un albero di ricerca usando gli operatori degli alberi binari.

Soluzione: Sezione 2.2

1.3 Concatenazione (Esercizio 6.6 del libro)

Dati due alberi binari di ricerca T_1 e T_2 tali che le chiavi in T_1 sono tutte minori delle chiavi in T_2, scrivere una procedura che restituisce un albero di ricerca contenente tutte le chiavi in tempo $O(h)$, dove h è l'altezza massima dei due alberi.

Soluzione: Sezione 2.3

1.4 Realizzazione di insiemi con alberi Red-Black (Esercizio 6.8 del libro)

Si assuma di rappresentare gli insiemi con alberi Red-Black. Quali sono le complessità delle operazioni union(), difference() e intersection()?

Soluzione: Sezione 2.4

1.5 Colorazione Red-Black (Esercizio 6.9 del libro)

Quali degli alberi in Fig. 1 possono essere colorati rispettando i vincoli degli alberi Red-Black, e quali no? Si assuma che tutti questi nodi siano interni, e che i nodi foglia **nil** non siano rappresentati.

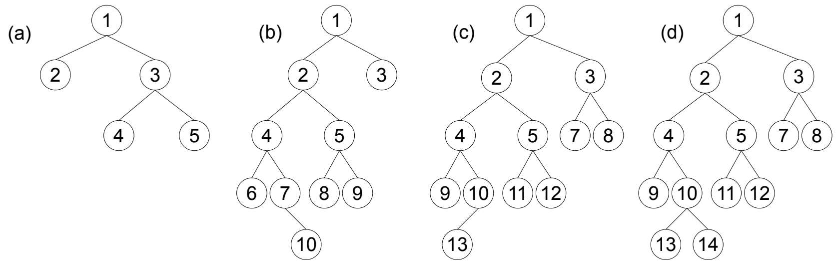

Figura 1: Alberi da colorare

Soluzione: Sezione 2.5

1.6 Differenze tra ABR e alberi *heap*

Qual è la differenza fra la proprietà degli ABR e la proprietà *min-heap*? È possibile utilizzare la seconda per elencare in modo ordinato le chiavi di un albero di n nodi in tempo $O(n)$?

Soluzione: Sezione 2.6

1.7 Proprietà degli ABR

Dimostrate che se un nodo in un ABR ha due figli, allora il suo successore non ha un figlio sinistro e il suo predecessore non ha un figlio destro.

Soluzione: Sezione 2.7

1.8 Distribuzione di valori in un albero binario di ricerca completo

Si consideri un albero binario completo T con 10 nodi. Etichettatare i nodi di T con i seguenti numeri: (12, 7, 32, 15, 23, 37, 18, 21, 17, 33), in modo tale che T sia un albero binario di ricerca.

Soluzione: Sezione 2.8

1.9 Distanza minima

Dati due numeri interi x ed y definiamo la distanza tra x ed y come $d(x, y) = |x - y|$. Sia T un albero binario di ricerca le cui chiavi sono numeri interi. Si descriva un algoritmo per determinare due elementi di T aventi distanza minima.

Soluzione: Sezione 2.9

2 Soluzioni

2.1 Versione ricorsiva di insertNode() (Esercizio 6.2 del libro)

La versione ricorsiva illustrata nell'algoritmo seguente ha ovviamente costo computazionale $O(h)$, dove h è l'altezza dell'albero. La chiamata iniziale è insertNodeRecursive(T, **nil**, x, v).

```
TREE insertNodeRecursive(TREE T, TREE p, ITEM x, ITEM v)
if T = nil then
    TREE n ← Tree(x, v)
    link(p, n, x)
    if p = nil then return n                      % Primo nodo ad essere inserito
else
    if x < T.key then
        insertNodeRecursive(T.left, T, x, v)
    else if x > T.key then
        insertNodeRecursive(T.right, T, x, v)
    else
        T.value ← v                               % Chiave già presente
return T                                          % Ritorna albero non modificato
```

2.2 Verifica ABR (Esercizio 6.3 del libro)

L'algoritmo deve confrontare il valore locale con tutti i valori dei sottoalberi destro e sinistro. Assumiamo, per semplicità di espressione, che ad ogni nodo t dell'albero siano associate due variabili t.*min* e t.*max*, che contengono rispettivamente il valore minimo e il valore massimo contenuto nel sottoalbero radicato in t. Il costo è quello di una visita, quindi $O(n)$.

```
boolean verifyABR(TREE t)
if t = nil then
    return true
if not (verifyABR(t.left) and verifyABR(t.right)) then
    return false
min ← t.value
max ← t.value
if t.left() ≠ nil then
    min ← min(min, t.left.min)
    max ← max(max, t.left.max)
if t.right() ≠ nil then
    min ← min(min, t.right.min)
    max ← max(max, t.right.max)
return (t.left ≠ nil or t.value > t.left.max) and (t.right ≠ nil or t.value < t.right.max)
```

2.3 Concatenazione (Esercizio 6.6 del libro)

Si cerca il massimo valore v contenuto in T_1, e si "attacca" la radice di T_2 come figlio destro di v, attraverso l'operazione link() definita nel libro. Si noti che v non può avere un figlio destro, altrimenti questo sarebbe il massimo. Il costo dell'operazione è dato dalla ricerca del minimo, che è $O(h)$ dove h è l'altezza massima fra i due alberi.

---

concatenate(TREE T_1, TREE T_2)

TREE $v \leftarrow$ max(T_1)
link($v, T_2, T_2.value$)

2.4 Realizzazione di insiemi con alberi Red-Black (Esercizio 6.8 del libro)

Detti n_1 e n_2 le dimensioni degli alberi da unire ed n la dimensione dell'albero risultante, gli algoritmi potrebbero essere realizzati nel modo seguente:

- union(): si effettua un'operazione di merge visitando iterativamente e in ordine entrambi gli alberi. Siano x_1 e x_2 i valori sotto esame dei due alberi: se $x_1 < x_2$, si avanza nell'albero T_1; se $x_1 > x_2$, si avanza nell'albero T_2; se invece sono uguali, si avanza in entrambi gli alberi. In tutti e tre i casi, il valore minimo viene inserito nell'albero unione. La prima coppia sotto esame è data dai due valori minimi.
- intersection(): si effettua un'operazione di merge visitando iterativamente e in ordine entrambi gli alberi. Siano x_1 e x_2 i valori sotto esame dei due alberi: se $x_1 < x_2$, si avanza nell'albero T_1; se $x_1 > x_2$, si avanza nell'albero T_2; se invece sono uguali, si inserisce il valore nell'albero intersezione e si avanza in entrambi gli alberi. La prima coppia sotto esame è data dai due valori minimi.
- difference(): si effettua un'operazione di merge visitando iterativamente e in ordine entrambi gli alberi. Siano x_1 e x_2 i valori sotto esame dei due alberi: se $x_1 < x_2$, si avanza nell'albero T_1 e si inserisce x_1 nell'albero differenza; se $x_1 > x_2$, si avanza nell'albero T_2; se invece sono uguali, si avanza in entrambi gli alberi senza inserire. La prima coppia sotto esame è data dai due valori minimi.

In tutti e tre i casi, l'algoritmo richiede di visitare, nel caso pessimo, tutti i nodi dei due alberi e di inserire, quando necessario, un valore nell'albero risultato; il costo è quindi $n_1 + n_2 + n \log n$.

2.5 Colorazione Red-Black (Esercizio 6.9 del libro)

L'albero a) può rispettare le proprietà, colorando di rosso il nodo 3 e di nero tutti gli altri. L'albero b) non può essere colorato, in quanto ha un cammino radice-foglia di 5 nodi e uno di 2 nodi. Il secondo può avere al massimo due nodi neri, mentre il secondo ne deve avere almeno 3, perché il primo nodo è nero e non ci possono essere due rossi in fila. Questa sarebbe una violazione della regola sull'altezza nera. Gli alberi c) e d) rispettano le regole, colorando di rosso i nodi 2, 13 e 14.

2.6 Differenze tra ABR e alberi *heap*

In un *min-heap*, la chiave di un nodo è più piccola di entrambe le chiavi dei suoi figli. In albero binario di ricerca, la chiave di un nodo è più grande o uguale alla chiave del suo figlio sinistro, e più piccola o uguale alla chiave del suo figlio destro. Entrambe le proprietà sono ricorsive. È importante notare che la proprietà di *heap* non ci aiuta a cercare un valore, perché non ci dice da che parte andare, visto che i valori sinistro e destro non hanno ordine particolare.
Se la proprietà di *heap* ci permettesse di visitare lo *heap* in tempo $O(n)$, visto che la costruzione di uno *heap* richiede tempo $O(n)$ avremmo risolto il problema dell'ordinamento in tempo lineare, cosa contraria al limite inferiore $\Omega(n \log n)$ per l'ordinamento basato su confronti.

2.7 Proprietà degli ABR

Discutiamo il caso del successore; il predecessore è simmetrico. Se un nodo ABR ha due figli, ha un figlio destro e quindi il suo successore si trova sicuramente nel sottoalbero destro e ne rappresenta il minimo. Se tale elemento avesse un figlio sinistro, questo sarebbe inferiore al minimo, il che è assurdo.

2.8 Distribuzione di valori in un albero binario di ricerca completo

Cenni per una soluzione generale: Data la forma dell'albero e gli n numeri da inserire, è possibile conoscere il numero di nodi che si trovano a destra e a sinistra della radice. Basta quindi ordinare la sequenza, e individuare il "perno" che fungerà da radice. Per esempio, supponendo che ci siano n_s nodi a sinistra e n_d nodi a destra, con $n_s + n_d + 1 = n$, si ottiene che la radice è il nodo $(n_s + 1)$-esimo. Si applica poi il meccanismo ricorsivamente ai sottoalberi destro e sinistro.

2.9 Distanza minima

Si effettua una in-visita dell'albero, ovvero si analizza l'albero seguendo l'ordine crescente; per ogni numero incontrato, si calcola la distanza con il valore precedente e la si confronta con il minimo trovato finora (opportunamente inizializzata al valore $+\infty$). Il costo è pari al costo di visita dell'albero, ovvero $O(n)$, dove n è il numero di nodi.

```
mindist(TREE t)
  TREE u ← t.min()
  integer min ← +∞
  integer prev ← −∞
  while u ≠ nil do
      if u.value − prev < min then
          min ← u.value − prev
      prev ← u.value u ← u.successorNode()
  return min
```

3 Problemi aperti

3.1 Rotazioni: conservazione delle proprietà di ordine

Dimostrare che gli algoritmi di rotazione in alberi binari di ricerca preservano sempre l'ordinamento inordine dei nodi di un albero binario.

3.2 Cancellazione

Dato in input un albero binario di ricerca T e due chiavi x, y presenti nell'albero, si provi (o si produca un controesempio) che cancellando prima x e poi y si ottiene lo stesso albero che si ottiene cancellando prima y e poi x.

3.3 Albero ABR di altezza minima

Sia dato un vettore A di n elementi ordinati in ordine crescente. (a) Scrivere un algoritmo ricorsivo (del tipo divide et impera) che costruisca l'albero binario di ricerca di altezza minima che ha per chiavi gli n elementi del vettore. (b) Scrivere l'equazione di ricorrenza per il tempo di esecuzione dell'algoritmo definito al passo precedente.