Introduzione agli Algoritmi - 9 giugno 2015
Prof. Emanuela Fachini (canale 1) e Prof. Irene Finocchi (canale 2)
Parte I

Esercizio 1

Date due funzioni f(n) = Ω(lg n) e g(n) = O(n), si considerino le seguenti affermazioni.

1. • g(n) = O(f (n))
2. • f (n) = Ω(lg g(n))
3. • f (n) + g(n) = O(n)

Per ciascuna affermazione si stabilisca se è sempre vera, sempre falsa, o se esistono casi in cui è vera e casi in cui è falsa. Per ogni affermazione né sempre vera né sempre falsa, si specifichino due funzioni f(n) e g(n) che la rendono vera e due che la rendono falsa.

Sol.
1. Se f(n) = lg n e g(n) = n l'affermazione è falsa.
Se f(n) = n (o una qualsiasi funzione che cresce più che linearmente) e g(n) = n l'affermazione è vera.
2. Questa affermazione è sempre vera infatti si sa per ipotesi che esistono n_0,c ≥0 tali che 0≤ g(n) ≤ cn, per n≥ n_0. Quindi lg g(n) ≤ lg cn = lg c + lg n; inoltre sempre per ipotesi esistono m_0,d ≥0 tali che f(n) ≥ d lg n, per n≥ m_0.
Ci si chiede se esistono n',k≥0 tali che f(n) ≥ k lg g(n), per ogni n≥ n', questo è vero se f(n) ≥ k (lg c + lg n) (che limita superiormente g(n)) e questo è vero prendendo k = d per ogni n ≥ max{n_0,m_0,}.
3. Se f(n) = n e g(n) = n l'affermazione è vera
Se f(n) = n^2 (o una qualsiasi funzione che cresce più in fretta) e g(n) = n l'affermazione è falsa.

Esercizio 2

Si determini il tempo di esecuzione nel caso peggiore T(n) della seguente funzione:

```
void f(int A[], int inizio, int fine) {
        int n = fine – inizio + 1
        sia B un array di n interi
        if (n>1) {
                copia(B, A, inizio, fine)
                f(A, inizio, inizio+n/3)
                f(A, inizio+2n/3+1, fine)
        }
}
```

dove copia(B,A,i,j) copia in B il contenuto dell'array A dalla posizione i alla posizione j.

Introduzione agli Algoritmi
9 giugno 2015
Prof. Emanuela Fachini (canale 1) e Prof. Irene Finocchi (canale 2)

Sol. La relazione di ricorrenza è $T(n) = 2T(n/3) + \Theta(n)$, perché, oltre alle istruzioni di costo costante c'è solo la copia dell'array in un altro che ha un costo lineare nel numero degli elementi; le due chiamate agiscono su 1/3 degli elementi (numero elementi prima chiamata: inizio + n/3 - inizio +1 = n/3+1, numero elementi seconda chiamata fine - inizio - 2n/3 - 1 +1 = n - 2/3n - 1 = 1/3n - 1).
La relazione da risolvere diventa:
$T(n) = 2T(n/3) + cn$, per $n \geq 2$
$T(n) = d$ altrimenti
L'albero della ricorsione è dunque:

cn ——————————————- cn

cn/3 cn/3 ——————————- 2cn/3

cn/9 cn/9 cn/9 cn/9 ——————— 4cn/9

Il prossimo livello sarà di 8 nodi e n sarà diviso ancora per 3 quindi è evidente che costo calcolato per il generico livello i è $(2/3)^i cn$. Poiché un somma finita di potenze con base minore di 1 è maggiorabile con la serie infinita che converge a una costante, possiamo concludere che la somma in questione è O(n), visto che il termine iniziale è una costante.
Con l'induzione possiamo verificare questa ipotesi.
Passo induttivo della prova che $T(n) \leq kn$, per un certo $k > 0$ e a partire da un certo n:
$T(n) = 2T(n/3) + cn \leq 2kn/3 + cn$ (applicando la definizione di T e l'ipotesi induttiva).
Si deve verificare se esiste un $k>0$ e un n_0 per cui $2kn/3 + cn \leq kn$, per ogni $n \geq n_0$.
$2kn/3 + cn \leq kn \Leftrightarrow 2k + 3c \leq 3k \Leftrightarrow 3c \leq k$, quindi la tesi è vera per $k \geq 3c$ e per ogni $n \geq 1$.
Per n=1 però $T(n) = d$, quindi possiamo dire che è vera per $n>1$, visto che interessa la crescita asintotica o prendere $k > \max\{d,3c\}$ e includere anche il caso di n=1.

Esercizio 3
Si consideri un array di n numeri inizialmente ordinato in ordine crescente, ma nel quale k valori sono stati diminuiti per cui risultano "fuori posto".

- Si scriva un algoritmo ***in loco*** che riordina l'array in tempo O(kn) nel caso peggiore.

- Si scriva un algoritmo alternativo, ***non*** necessariamente ***in loco***, che riordina l'array in tempo O(n+k lg k) nel caso peggiore.

Sol. Per il punto 1, basta osservare che nel caso specifico l'insertion sort ha la complessità richiesta. Infatti i "fuori posto" vengono inseriti con una complessità lineare nel numero degli elementi da esaminare, che al più è n.

Per il punto 2, l'idea è di copiare in un array d'appoggio i k elementi fuori posto, per poi ordinarli con il mergesort, con un costo θ(k lg k) e infine fondere l'array ordinato così ottenuto con gli elementi rimanenti dell'array di partenza, questa operazione è lineare nel numero degli elementi da fondere, O(n), quindi si ha un costo O(n + k lg k).
Qualche dettaglio in più è necessario per realizzare l'idea.
Primo, come si individuano gli elementi fuori posto in un array A come quello descritto nella traccia?
La prima condizione cui si pensa è che è fuori posto ogni elemento A[i] tale che A[i-1] > A[i], visto che l'array è ordinato in ordine crescente, che è corretta se i k elementi diminuiti non sono consecutivi. Infatti se A[j] e A[j-1] sono stati diminuiti entrambi può capitare che A[j-1] ≤ A[j] ma che in realtà siano entrambi fuori posto:
per esempio se A all'inizio è A= [15,18,60,80,100] e si diminuiscono 3 elementi portando 18 a 9, 60 a 10 e 100 a 5, si ha l'array A= [15,9,10,80,5], e il confronto a due a due porterebbe a individuare come fuori posto solo 9 e 5, mentre anche 10 deve essere riposizionato nell'array ordinato finale.
Naturalmente può capitare che siano più di due gli elementi fuori posto consecutivi, quindi la soluzione è di portarsi dietro il più grande elemento precedente nella scansione, ogni elemento più piccolo o uguale di quello trovato tra gli elementi successivi viene considerato fuori posto, e quindi memorizzato in un array B, mentre il successivo elemento più grande diventa il nuovo elemento di confronto.
A questo punto si ha il vettore degli elementi fuori posto e si può ordinarlo con il mergesort. Questo array avrà dimensione al più k, perché qualche elemento diminuito potrebbe non essere fuori posto.
A livello implementativo bisognerà gestire questa situazione.
Come realizzare la fusione? Il modo più semplice e anche meno oneroso in termini di memoria aggiuntiva è di copiare in un array C, di al più n - k elementi, gli elementi non fuori posto durante la scansione precedentemente descritta. L'array C risulterà ordinato.
A questo punto si realizza la fusione in A dei due arrays B e C, con un costo O(n) sia in tempo che in spazio, perché il mergesort necessita di O(k) spazio per l'array di appoggio e l'array B, mentre la creazione di C impegna spazio O(n-k).

Se non si fa la copia in C degli elementi non fuori posto, una fusione è sempre possibile, ma si deve tener conto del fatto che in A ci sono ancora gli elementi fuori posto, quindi bisogna modificare l'operazione di fusione in modo tale da non copiare nel nuovo array, diciamo D, gli elementi fuori posto da A, visto che vengono presi da B. Se non ci sono duplicati si potrebbe, per ogni elemento preso da A, utilizzare la ricerca binaria in B per stabilire se l'elemento preso per il confronto è in realtà un fuori posto e quindi da trascurare. Questo comporta una complessità di tempo O(n lg k) e una complessità di spazio O(n+k). Se ci sono duplicati questo non va bene perché un duplicato potrebbe essere stato ottenuto per diminuzione e l'altro no, quindi uno starebbe al posto giusto.
Si può allora modificare la fusione "incorporando" il codice per la individuazione di un fuori posto e quindi confrontando ogni elemento in A con il massimo dei precedenti, e trascurandolo ai fini della fusione se risulta minore. Questa soluzione ha un costo asintotico di tempo equivalente alla più semplice in cui si crea l'array C, ma è chiaramente meno efficiente; per quanto riguarda lo spazio, l'occupazione è O(n+k). Anche qui asintoticamente l'occupazione è O(n) perché $k \leq n$, ma si crea un nuovo array di al più k elementi e un nuovo array di n elementi, mentre nella prima soluzione si crea un array di al più k elementi e uno di al più n-k, realizzando la fusione nell'array di partenza.

Parte II

Esercizio 4

Si dimostri che nel modello basato su confronti sono necessari $\Omega(n \lg n)$ confronti per ordinare n elementi arbitrari.

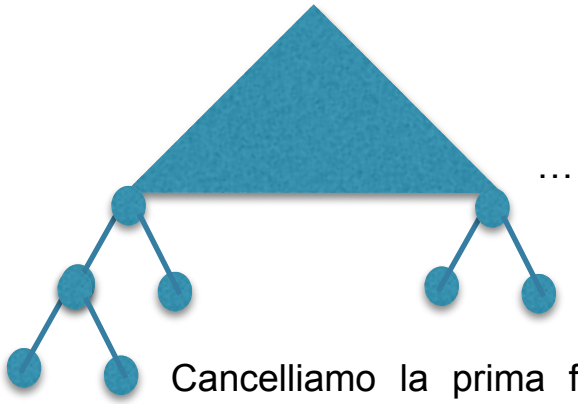
Sol. Si veda libro di testo e lucidi.

Esercizio 5

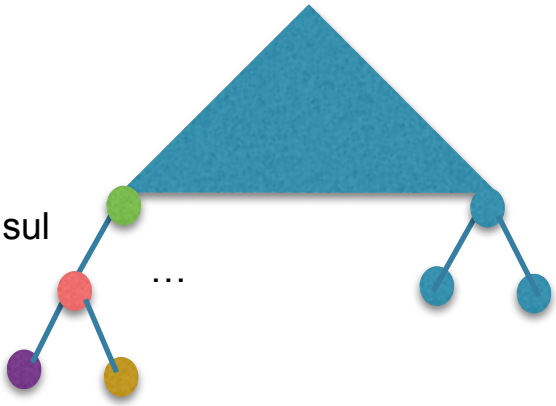
1. Si consideri un AVL completo fino al penultimo livello e avente nell'ultimo livello solo due foglie posizionate il più a sinistra possibile. Si dimostri che se si cancella la prima foglia sul penultimo livello (da sinistra) basta una sola rotazione per ribilanciare l'albero.

Sol.
Sia T come descritto:

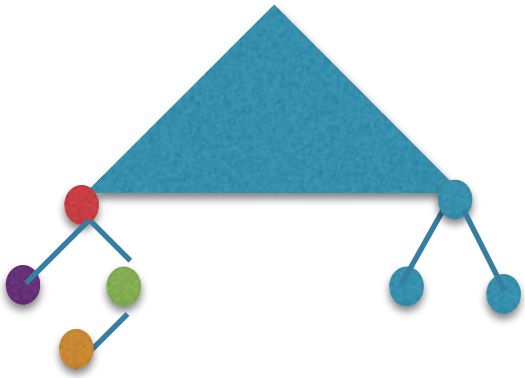
Introduzione agli Algoritmi - 9 giugno 2015
Prof. Emanuela Fachini (canale 1) e Prof. Irene Finocchi (canale 2)



Cancelliamo la prima foglia sul penultimo livello:

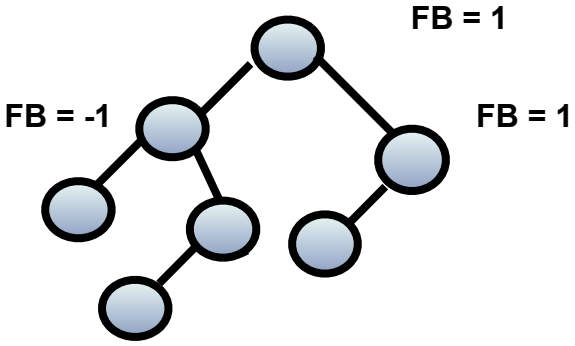


Eseguiamo una rotazione a destra sul primo nodo del terz'ultimolivello e otteniamo

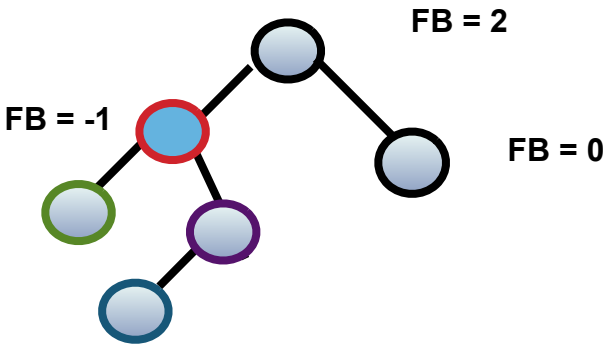


Il sotto albero radicato nel nodo rosso dopo la rotazione ha la **stessa altezza** di quello originario prima della cancellazione e quindi non sono necessarie altre rotazioni.

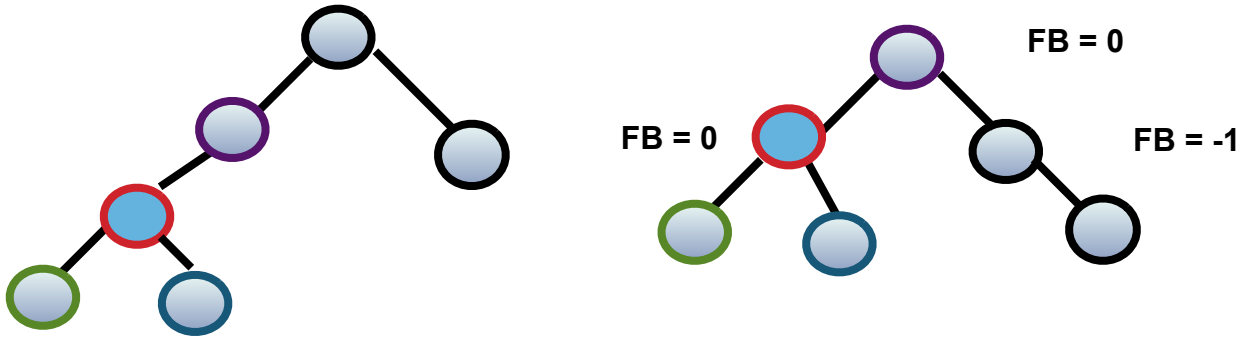
2. Si disegni un albero AVL con 7 nodi in cui la cancellazione di un nodo comporta una rotazione doppia; si disegni l'albero prima e dopo la cancellazione e la rotazione.



Introduzione agli Algoritmi
9 giugno 2015
Prof. Emanuela Fachini (canale 1) e Prof. Irene Finocchi (canale 2)



Poiché il nodo che ha FB illegale, 2, ha il figlio sinistro con FB discorde, -1 si deve fare una doppia rotazione:



Esercizio 6

Si consideri la classe ABRPS di Alberi Binari di Ricerca Pesanti a Sinistra, ovvero alberi binari di ricerca in cui per ogni nodo il numero dei nodi nel suo sottoalbero sinistro è maggiore o uguale a quello dei nodi nel suo sottoalbero destro.
Ogni ABRPS T è rappresentato in memoria con strutture a puntatori e ogni nodo v ha un campo aggiuntivo *size* che contiene il numero dei nodi nel sottoalbero radicato in v.

Si scriva un algoritmo per l'inserimento di un nuovo elemento in un ABRPS, utilizzando come operazione di base il normale inserimento in un ABR e mettendo in evidenza quali operazioni sono necessarie sull'albero per ripristinare la proprietà di essere pesante a sinistra.

Si dimostri la correttezza delle operazioni di ripristino, in modo analogo a come si è fatto nel caso degli AVL.

Sol. Si usa la funzione di inserimento in un ABR. A questo punto bisogna modificare i campi size dei nodi lungo il cammino dal padre del nuovo nodo inserito fino alla radice e controllare che la condizione di pesantezza a sinistra sia ancora verificata. Se non lo fosse una rotazione a sinistra sul nodo in cui la proprietà è violata ripristina la proprietà **perché almeno un nodo certamente passa dal sotto albero destro al sinistro.** Può accadere che si debbano eseguire h rotazioni, dove h è l'altezza, per esempio se l'albero è completo e inseriamo un nuovo nodo come foglia più a destra, la complessità di tempo asintotica è dunque O(h).
Usiamo la funzione insertABR(T,x), dove x è il nodo da aggiungere, con i campi opportunamente inizializzati, in particolare con il campo size a 1, che inserisce un nuovo nodo in un ABR e restituisce il puntatore al padre di x.
la funzione leftRot(P), che esegue la rotazione a sinistra su P, senza modificare il valore del puntatore P e una funzione aggiornaSizeLRot(P) che aggiorna i campi size dei nodi coinvolti nella rotazione a sinistra.
Una possibile implementazione in pseudocodice:

```
InsertABRPS(T,x)
 y = insertABR(T,x)
 while (y ≠ NULL)
    { if (y.left ≠ NULL) a = y.left.size else a=0
      if (y.right≠NULL) b = y.right.size else b=0
        y.size = a+b +1
(aggiornamento dovuto all'inserimento, il +1 è per il nodo y  stesso)
        if (a < b)
 (verifica pesantezza a sinistra, se a<b il puntatore al figlio destro non è
nullo e quindi si può applicare la rotazione a sinistra)
             { aggiornaSizeLRot(y)
              leftRot(y)
              y=y.p.p} (risalita al nonno di y)
        y= y.p}
```

E quello per l'aggiornamento dei campi size in vista della rotazione:

Introduzione agli Algoritmi
9 giugno 2015
Prof. Emanuela Fachini (canale 1) e Prof. Irene Finocchi (canale 2)

```
aggiornaSizeLRot(y)
   if (y.left ≠ NULL) a = y.left.size  else  a = 0
   if (y.right.left ≠ NULL) b = y.right.left.size  else b = 0
  (y.left e y.right.left saranno i due sottoalberi di y, dopo la rotazione)
   y.right.size = y.size
   (y.right sarà la nuova radice del sotto albero radicato in y)
   y.size = a+b +1
```