Algoritmi e Strutture Dati
21 febbraio 2023

Note

1. La leggibilità è un prerequisito: parti difficili da leggere potranno essere ignorate.
2. Quando si presenta un algoritmo è fondamentale spiegare l'idea e motivarne la correttezza.
3. L'efficienza e l'aderenza alla traccia sono criteri di valutazione delle soluzioni proposte.
4. Si consegnano tutti i fogli, con nome, cognome, matricola e l'indicazione *bella copia* o *brutta copia*.

Domande

Domanda A (6 punti) Dare la definizione formale delle classi $O(f(n))$ e $\Omega(f(n))$ per una funzione $f(n)$. Mostrare che se $f(n) = O(n^2)$ e $g(n) = \Omega(n)$, con $g(n) > 0$ per ogni n, allora $f(n)/g(n) = O(n)$.

Soluzione: La classi $O(f(n))$ e $\Omega(g(n))$ sono definite come:

$$O(f(n)) = \{g(n) : \exists c > 0.\ \exists n_0.\ \forall n \geq n_0.\ 0 \leq g(n) \leq cf(n)\}$$
$$\Omega(f(n)) = \{g(n) : \exists d > 0.\ \exists n_0.\ \forall n \geq n_0.\ 0 \leq d\, f(n) \leq g(n)\}$$

Si assuma che $f(n) = O(n^2)$ e $g(n) = \Omega(n)$. Ovvero, esistono $c > 0, n_0$ tali che per ogni $n \geq n_0$

$$0 \leq f(n) \leq cn^2$$

e $d > 0, m_0$ tali che per ogni $n \geq m_0$

$$0 < dn \leq g(n)$$

Quindi, per $n \geq \max\{n_0, m_0\}$ si ha che

$$\begin{aligned} 0 &\leq f(n)/g(n) & \\ &\leq cn^2/g(n) & [\text{poiché } f(n) \leq cn^2] \\ &\leq cn^2/(dn) & [\text{poiché } g(n) \geq dn] \\ &= (c/d)n. & \end{aligned}$$

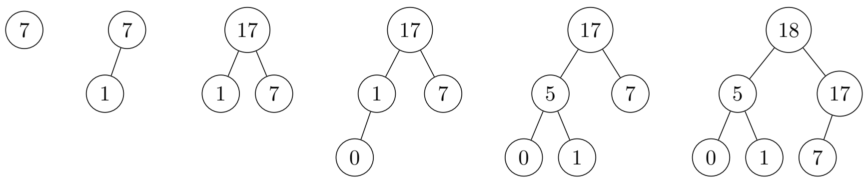
dato che $c/d > 0$ questo conclude la prova.

Domanda B (7 punti) Dare la definizione di max-heap. Data la sequenza di elementi `7, 1, 17, 0, 5, 18,` si specifichi il max-heap ottenuto inserendo, a partire da uno heap vuoto, uno alla volta questi elementi nell'ordine indicato e infine rimuovendo 0. Si descriva sinteticamente come si procede per arrivare al risultato.

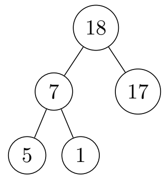
Soluzione: Un max-heap è un albero binario ordinato quasi-completo (tutti i livelli completi, a parte l'ultimo, nel quale le foglie devono essere "addossate" a sinostra) con la proprietà che per ogni nodo x, se x non è radice, gli antenati di x hanno chiave maggiore o uguale a quella di x, o equivalentemente ogni nodo x ha chiave maggiore o uguale a quella dei suoi discendenti.

Si procede inserendo nel min-heap i vari elementi con la procedura `HeapInsert` a partire da heap vuoto. La procedura inserisce l'elemento come prima foglia utile (ultimo elemento dell'array) e richiama la procedura `MaxHeapifyUp` per ripristinare la proprietà di min-heap (scambiando il nodo con il suo genitore finché la chiave di questo è inferiore a quella del figlio).

Gli inserimenti nell'ordine producono gli heap indicati in figura



Infine, la rimozione di un nodo con chiave x si realizza sostituendolo con l'ultima foglia y e richiamando `MaxHeapifyUp`, se $y > x$ oppure `MaxHeapify`, se $y < x$. Nello specifico per eliminare 0 lo si rimpiazza con l'ultima foglia 7 e si richiama `Max`, ottenendo il min-heap



che in forma di array è [12, 19, 15, 20, 21, 22].

Esercizi

Esercizio 1 (10 punti) Realizzare un arricchimento degli alberi binari di ricerca che permetta di ottenere per ogni nodo x, il grado di bilanciamento del sottoalbero radicato in x, definito come $h_x/\log_2(n_x+1)$ dove h_x e n_x indicano rispettivamente l'altezza e il numero di nodi del sottoalbero radicato in x (si intende che un albero costituito da un solo nodo abbia altezza 1).

Indicare quali campi occorre aggiungere ai nodi. Fornire lo pseudo-codice per la funzione `bal(x)` che restituisce il grado di bilanciamento del sottoalbero radicato in x e la procedura di inserimento di un nodo `insert(T,z)`. Valutare la complessità delle funzioni definite.

Soluzione: L'idea è quella di arricchire la struttura facendo in modo che ogni nodo x, oltre ai campi usuali, abbia due campi aggiuntivi: x.`h` che contiene l'altezza del sottoalbero radicato in x e x.`size` che contiene il numero dei nodi in tale sottoalbero.

A questo punto è immediato realizzare la funzione `bal(x)` di tempo costante

```
bal(x)
  if x <> nil
     return x.h/log_2(x.size)
  else
     error
```

Invece per l'inserimento, si modifica la procedura standard, che discende l'albero dalla radice fino alla posizione in cui inserire il nuovo nodo, facendo in modo che tutti i nodi attraversati e che quindi conterranno nel sottoalbero il nuovo nodo, abbiano il campo `size` incrementato di 1. Inoltre, una eventuale modifica dell'altezza va propagata verso l'alto. Questo accade quando il parent del nodo inserito era una foglia e quindi la sua altezza, prima dell'inserimento, era 1.

```
Insert(T,z)
   y = nil
   x = T.root

   while (x <> nil)
      y = x
      x.size++
      if (z.key < x.key)
         x = x.left
      else
         x = x.right

   // inizializza i campi del nodo z
   z.size = 1
   z.left = z.right = nil
   z.h = 1

   z.p = y
   if (y <> nil)      // se z non radice
      if (z.key < y.key)
         y.left =  z
      else
         y.right = z

      // se y era una foglia, ovvero se l'altezza di y era 1, la sua
      // altezza e conseguementemente quella degli antenati va
      // aumentata di 1 risalendo i nodi attraversati nella discesa
      if y.h == 1
         while y <> nil
               y.h ++
               y = y.p
```

La complessità resta $O(h)$ dove h è l'altezza dell'albero.

Esercizio 2 (9 punti) Per $n > 0$, siano dati due vettori a componenti intere $\mathbf{a}, \mathbf{b} \in \mathbf{Z}^n$. Si consideri la quantità $c(i,j)$, con $0 \le i \le j \le n-1$, definita come segue:

$$c(i,j) = \begin{cases} a_i & \text{se } 0 < i \le n-1 \text{ e } j = n-1, \\ b_j & \text{se } i = 0 \text{ e } 0 \le j \le n-1, \\ c(i-1,j-1) \cdot c(i,j+1) & 0 < i \le j < n-1. \end{cases}$$

Si vuole calcolare la quantità $m = \max\{c(i,j) : 0 \le i \le j \le n-1\}$.

(a) Fornire un algoritmo iterativo bottom-up per il calcolo di m.

(b) Valutare la complessità *esatta* dell'algoritmo, associando costo unitario alle moltiplicazioni tra numeri interi e costo nullo a tutte le altre operazioni.

Soluzione:

(a) Date le dipendenze tra gli indici nella ricorrenza, un modo corretto di riempire la tabella è attraverso una scansione in cui calcoliamo gli elementi in ordine crescente di indice di riga e, per ogni riga, in ordine decrescente di indice di colonna. Il codice è il seguente.

```
COMPUTE(a,b)
n <- length(a)
m = -infinito
for i=1 to n-1 do
   C[i,n-1] <- a_i
   m <- MAX(m,C[i,n-1])
for j=0 to n-1 do
   C[0,j] <- b_j
   m <- MAX(m,C[0,j])
for i=1 to n-2 do
   for j=n-2 downto i do
      C[i,j] <- C[i-1,j-1] * C[i,j+1]
      m <- MAX(m,C[i,j])
return m
```

(b)

$$T(n) = \sum_{i=1}^{n-2} \sum_{j=i}^{n-2} 1 = \sum_{i=1}^{n-2} (n-1-i) = \sum_{k=1}^{n-2} k = (n-1)(n-2)/2.$$