DIPARTIMENTO DI INGEGNERIA ELETTRICA E DELLE TECNOLOGIE DELL'INFORMAZIONE
UNIVERSITÀ DEGLI STUDI DI NAPOLI FEDERICO II

Corso di Laurea Magistrale in Ingegneria Informatica

CORSO DI ALGORITHMS AND DATA STRUCTURES

Prof. ROBERTO PIETRANTUONO

Homework set #1

Istruzioni

Si prepari un file PDF riportante il vostro nome e cognome (massimo 2 studenti). Quando è richiesto di fornire un algoritmo, si alleghi un file editabile (ad esempio, .txt, .doc) riportante l'algoritmo in un linguaggio a scelta, corredato da almeno tre casi di test e dall'indicazione della complessità. Laddove opportuno, si fornisca una breve descrizione della soluzione: l'obiettivo non è solo eseguire l'esercizio e riportare il risultato, ma far comprendere lo svolgimento.

Esercizio 1.1. Notazione asintotica
Per ognuna delle seguenti affermazioni, si dica se essa è **sempre vera**, **mai vera**, o **a volte vera,** per funzioni asintoticamente non-negative. Se la si considera sempre vera o mai vera, si spieghi il perché. Se è a volte vera, si dia un esempio per cui è vera e uno per cui è falsa.

- $f(n) = O(f(n)^2)$
- $f(n) + O(f(n)) = \Theta(f(n))$
- $f(n) = \Omega(g(n))$ e $f(n) = o(g(n))$ - Nota la notazione *little-o*

Esercizio 1.2. Complessità.
E' vero che $2^{n+1} = O(2^n)$? E' vero che $2^{2n} = O(2^n)$?

Esercizio 1.3. Ricorrenze
Fornire il limite inferiore e superiore per $T(n)$ nelle seguenti ricorrenze. Si assume che T(n) è costante per $n \leq 10$. Si fornisca il limite più stretto possibile giustificando la risposta (es: se T(n) è O(log$n$) essa è chiaramente anche O(n): la risposta dovrebbe essere O(log$n$)).

- $T(n)=2T(n/3)+n\lg n$
- $T(n) = 3T(n/5) + \lg^2 n$

Esercizio 1.5 Ricorrenze
Utilizzando l'albero di ricorsione, dimostrate che la soluzione della ricorrenza $T(n) = T(n/3) + T(2n/3) + cn$, dove c è una costante, è $\Omega(n\log n)$

Problema 1.1
Un array è detto ***unimodale*** se è formato da una sequenza crescente seguita da una decrescente; più precisamente, se c'è un indice $m \in \{1, 2, \ldots, n\}$ tale che:

- $A[i] < A[i] + 1$, per ogni $1 \leq i < m$, e
- $A[i] > A[i] + 1$, per ogni $m \leq i < n$.

A[m] è l'elemento massimo, ed è l'unico elemento "massimo locale" circondato elementi più piccoli (A[m -1] e A[m+1]).

Si fornisca un algoritmo per calcolare il massimo elemento di un array unimodale A[1, … n] che esegue in O(lg*n*). Si dimostri che la complessità è O(lg*n*). (Suggerimento: esiste una soluzione molto simile alla ricerca binaria)

Problema 1.2
Dato un insieme di stringhe, si implementi un algoritmo *divide et impera* per trovare il prefisso in comune più lungo.

Si alleghi al PDF un file editabile riportante l'implementazione in un linguaggio a scelta, corredato da almeno tre casi di test oltre quello di esempio riportato di seguito. Si riporti anche l'analisi di complessità.

Esempio

Stringhe di input:
apple
ape
april
applied

Output:
ap

Problema 1.3
Descrizione.
Se inseriamo un insieme di *n* elementi in un albero di ricerca binario (*binary search tree*, BST) utilizzando TREE-INSERT, l'albero risultante potrebbe essere molto sbilanciato. Tuttavia, ci si aspetta che i BST costruiti casualmente siano bilanciati (ossia ha un'altezza attesa O(lg n)). Pertanto, se vogliamo costruire un BST con altezza attesa O(lg n) per un insieme fisso di elementi, potremmo permutare casualmente gli elementi e quindi inserirli in quell'ordine nell'albero.
Cosa succede se non abbiamo tutti gli elementi a disposizione in una sola volta? Se riceviamo gli elementi uno alla volta, possiamo ancora costruire casualmente un albero di ricerca binario da essi? Nel seguito è proposta una struttura dati che risponde affermativamente a questa domanda. Un ***treap*** è un albero binario di ricerca che usa una strategia diversa per ordinare i nodi. Ogni elemento x nell'albero ha una chiave *key*[x]. Inoltre, assegniamo *priority*[x], che è un numero casuale scelto indipendentemente per ogni x. Assumiamo che tutte le priorità siano distinte e anche che tutte le chiavi siano distinte. I nodi del *treap* sono ordinati in modo che (1) le chiavi obbediscano alla proprietà del *binary search tree* e (2) le priorità obbediscano alla proprietà *min-heap order* dell'heap. In altre parole:
- se *v* è un figlio sinistro di *u*, allora *key*[*v*] < *key*[*u*];
- se *v* è un figlio destro di *u*, allora *key*[*v*] > *key*[*u*]; e
- se *v* è un figlio di *u*, allora *priority*(*v*) > *priority*(*u*).

(Questa combinazione di proprietà è il motivo per cui l'albero è chiamato "***treap***": ha caratteristiche sia di un albero di ricerca binario che di un *heap*)

È utile pensare ai *treaps* in questo modo: supponiamo di inserire i nodi $x_1, x_2, \ldots x_n$, ciascuno con una chiave associata, in un *treap* in ordine arbitrario. Quindi il *treap* risultante è l'albero che si sarebbe formato se i nodi fossero stati inseriti in un normale albero binario di ricerca nell'ordine dato dalle loro priorità (scelte casualmente). In altre parole, *priority*[x_i] < *priority*[x_j] significa che x_i è effettivamente inserito prima di x_j.

Per inserire un nuovo nodo x in un *treap* esistente, si assegna dapprima ad x una priorità casuale *priority*[x]. Quindi si chiama l'algoritmo di inserimento, che chiameremo TREAP-INSERT, il cui funzionamento è illustrato nella Figura 1.

<u>Quesito</u>. **Fornire il codice della procedura TREAP-INSERT** in un linguaggio a scelta, allegando un file editabile**.** *Suggerimento: effettuare il consueto inserimento del BST, ed eseguire le rotazioni per ripristinare la proprietà del min-heap* (*min-heap order*)*).*

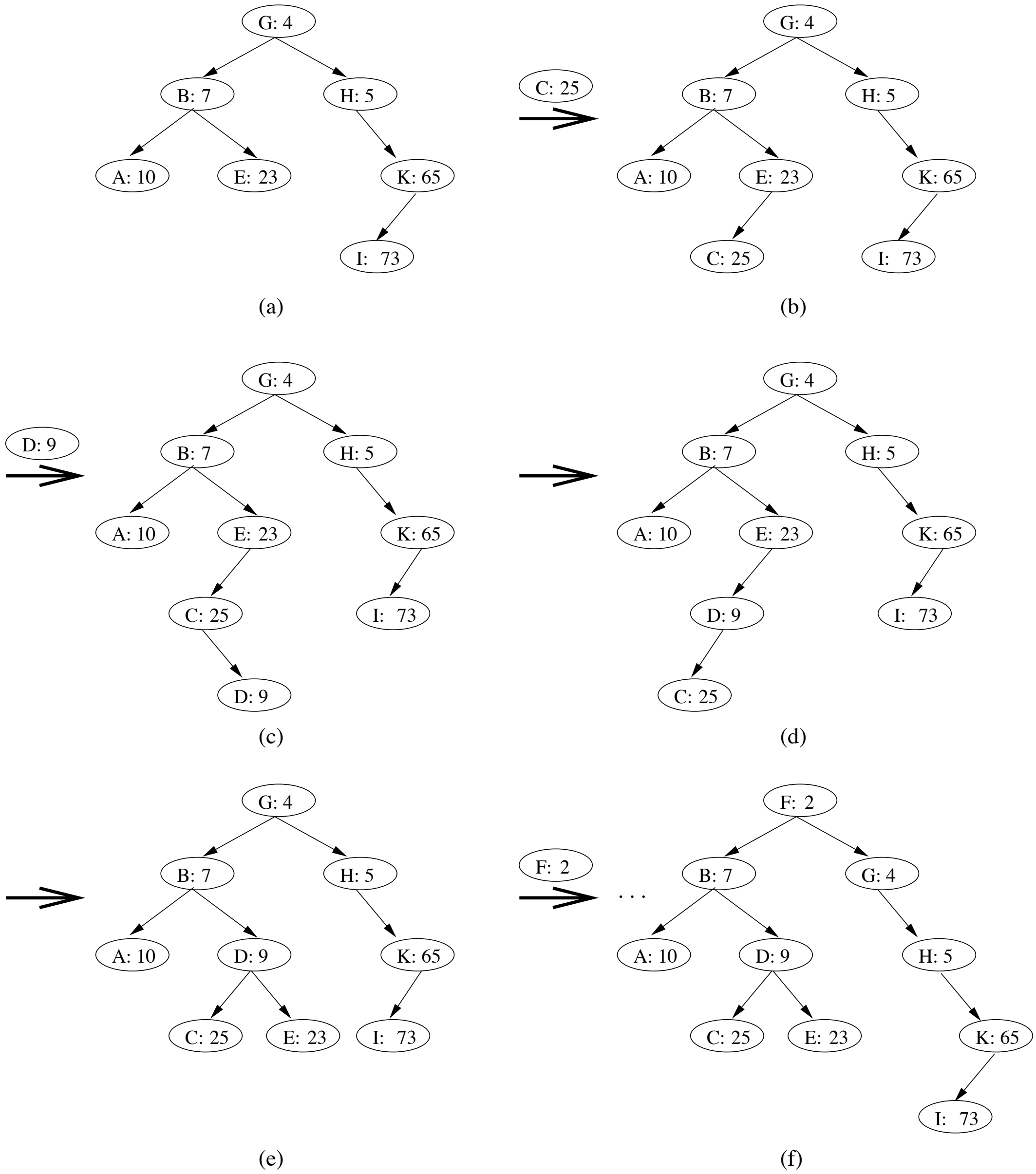

Figura 1. Operazioni di TREAP-INSERT. Ogni nodo è etichettato con key[x] : priority[x]. a) *Treap* prima dell'inserimento; b) *Treap* dopo aver inserito un nodo con chiave C e priorità 25; c-d) stadi intermedi quando si inserisce D (priorità 9); e) *Treap* dopo il completamento dell'inserimento delle parti c)-d); f) *Treap* dopo l'inserimento di F (priorità 2)