DIETI — Università degli Studi di Napoli Federico II
DIPARTIMENTO DI INGEGNERIA ELETTRICA E DELLE TECNOLOGIE DELL'INFORMAZIONE

Corso di Laurea Magistrale in Ingegneria Informatica

CORSO DI ALGORITMI E STRUTTURE DATI

Prof. ROBERTO PIETRANTUONO

Homeworks set #3

Istruzioni

Si prepari un file PDF riportante il vostro nome e cognome (massimo 2 studenti). Quando è richiesto di fornire un algoritmo, si intende scritto in pseudo-codice e riportato nel PDF. 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.

Homeworks esercitativi

Esercizio 3.1. Supponete di usare una funzione hash h per inserire n chiavi distinte in un array T di lunghezza m. Nell'ipotesi di *hashing* uniforme semplice, qual è il numero atteso di collisioni? Più precisamente, qual è la cardinalità attesa di $\{\{k, l\}: k \neq l \text{ e } h(k) = h(l)\}$?

Esercizio 3.2. Possiamo ordinare un dato insieme di n numeri costruendo prima un albero binario di ricerca che contiene tali numeri (usando ripetutamente TREE-INSERT per inserire i numeri uno alla volta) e stampando poi i numeri con un attraversamento simmetrico dell'albero. Quali sono i tempi di esecuzione nel caso peggiore e nel caso migliore per questo algoritmo di ordinamento?

Homeworks di verifica

Homework 3.1. Radix tree

Date due stringhe $a = a_0a_1\ldots a_p$ e $b = b_0b_1\ldots b_q$, dove a_i e b_j appartengono a un insieme ordinato di caratteri, diciamo che la stringa a è lessicograficamente minore della stringa b se è soddisfatta una delle seguenti condizioni:

1. Esiste un numero intero j, con $0 \leq j \leq min(p, q)$, tale che $a_i = b_i$, per ogni $i = 0, 1, \ldots j-1$ e $a_j < b_j$.
2. $p<q$ e $a_i = b_i$, per ogni $i = 0, 1, \ldots, p$.

Per esempio, se a e b sono stringhe di bit, allora 10100 < 10110 per la regola 1 (ponendo $j = 3$) e 10100 < 101000 per la regola 2. Questo è simile all'ordinamento utilizzato nei dizionari.

La struttura dati ***radix tree*** illustrata in Figura 1 contiene le stringhe di bit 1011, 10, 011, 100, e 0. Durante la ricerca di una chiave $a = a_0a_1\ldots a_p$ si va sinistra in un nodo di profondità i se $a_i = 0$, a destra se $a_i = 1$. Sia S un insieme di stringhe binarie distinte, la cui somma delle lunghezze è n. Dimostrate come utilizzare un ***radix tree*** per ordinare lessicograficamente le stringhe di S nel tempo $\Theta(n)$. Per l'esempio illustrato in figura, l'output dell'ordinamento dovrebbe essere la sequenza 0, 011, 10, 100, 1011.

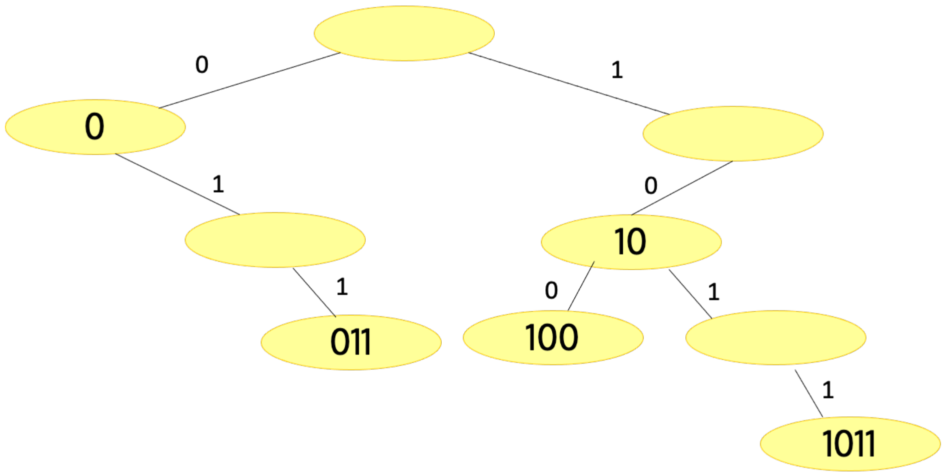

Figura 1. *Radix tree* con 1011, 10, 011, 100, e 0. Una chiave può essere determinata seguendo il cammino semplice dalla radice al nodo. I nodi vuoti sono presenti soltanto per stabilire un percorso per gli altri nodi.

Homework 3.2. Treaps

<u>*Descrizione.*</u>

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 <u>***treap***</u> 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, $\dots x_n$, 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 2.

Quesito. **Fornire lo pseudocodice della procedura TREAP-INSERT.** *Suggerimento: effettuare il consueto inserimento del BST, ed eseguire le rotazioni per ripristinare la proprietà del min-heap* (*min-heap order*)*).*

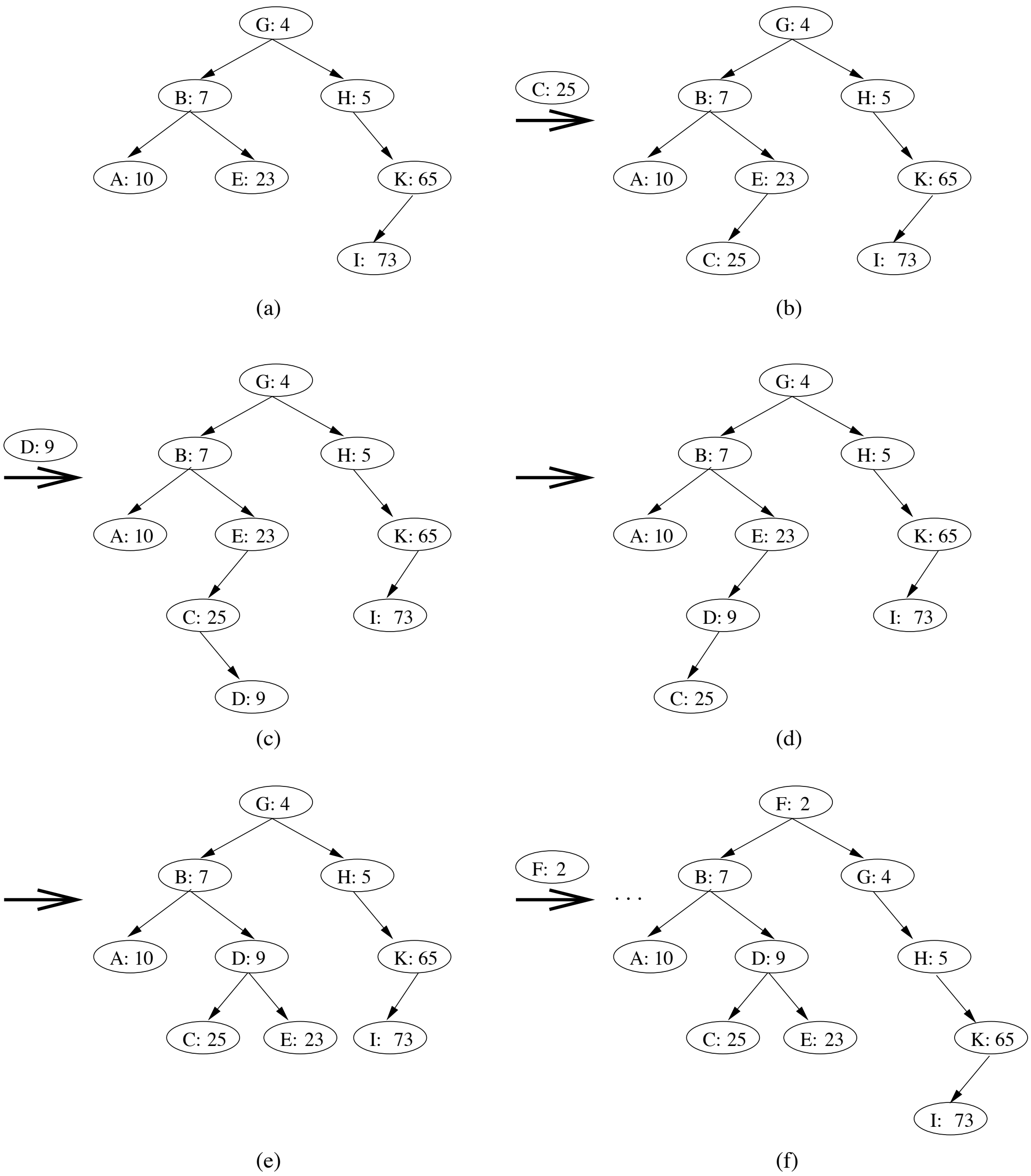

Figura 2. Operazioni di TREAP-INSERT. Ogni nodo è etichettato con key[x] : priority[x]. a) *Treap* prima dell'inserimento; b) *Treap* dopo aver inserito un nodo con chiave C e priorità 25; c-d) stadi intermedi quando si inserisce D (priorità 9); e) *Treap* dopo il completamento dell'inserimento delle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