Terzo Esonero

corso di laurea in **Matematica**

Informatica Generale, Esonero **3**

Ivano Salvo

SAPIENZA
UNIVERSITÀ DI ROMA

Terzo Esonero, 13/5/2022

Esercizio: Considerare il problema della *k*-mediana sull'insieme rappresentato da un Albero binario di ricerca. Sviluppare soluzioni seguendo le seguenti tracce:

1. **c**aricare le chiavi dell'ABR su una **coda** in un ordine opportuno, e poi usare la coda per calcolare la *k*-mediana.
2. **c**aricare le chiavi dell'ABR su una **pila** in un ordine opportuno, e poi usare la pila per calcolare la *k*-mediana.
3. **c**alcolare la k-mediana, visitando l'albero ricorsivamente ma **senza allocare strutture dati ausiliarie** (eventualmente usando funzioni ausiliarie con più parametri e/o valori di ritorno.
4. **aggiungere agli ABR** un'**ulteriore informazione** **su ciascun nodo**, in modo da poter calcolare la *k*-mediana in tempo $\Theta(h)$. Eventualmente, scrivere anche le funzioni `insABR` e `removeABR` che mantengono questa informazione coerente a seguito di inserimenti/rimozioni sull'albero.

Argomentare **brevemente** sulla **correttezza** e **complessità** computazionale delle funzioni scritte. La complessità va scritta in funzione di n (numero dei nodi dell'ABR) e/o h (l'altezza) e k (mediana da calcolare).

Nota sulla valutazione: Gli esercizi sono di difficoltà crescente. **2** esercizi buoni **sono sufficienti**, ma non è uguale fare **1+2** (~7pt) oppure **3+4** (~**10**pt).

Soluzioni Terzo Esonero

corso di laurea in **Matematica**

Informatica Generale, Lezione **24.3**

Ivano Salvo

SAPIENZA
UNIVERSITÀ DI ROMA

k-mediana & ABR

Esercizio Si consideri il problema di calcolare la k-mediana dell'insieme rappresentato da un *albero binario di ricerca*.

Dare degli algoritmi e valutarne la complessità in tempo e spazio nei seguenti scenari, in funzione di n (numero dei nodi dell'albero) e/o h (altezza dell'albero) e/o k (mediana da trovare).

Tutti gli esercizi chiedevano di trovare un algoritmo per calcolare la ***k*-mediana** in un **albero binario di ricerca**.

L'osservazione chiave è che in un ABR, **una visita in-order restituisce le chiavi in ordine ascendente**, situazione favorevole per calcolare la k-mediana.

Questa proprietà era stata sfruttata (in modo analogo) per determinare il **minimo intero mancante** (vedi Esercitazione **6**, Lezione **18**).

caso 1: coda

1. Il vostro capo, per reconditi motivi, vi obbliga a chiamare nella funzione `kMedianABR` una funzione `abrSuCoda(B, Q)` che visita l'albero B raccogliendo informazioni nella coda Q, e una funzione `m = kMedianaCoda(Q, k)` che usa opportunamente le informazioni memorizzate nella coda `Q`. Scrivere lo pseudocodice di `kMedianABR`, `abrSuCoda(B, Q)` e `kMedianaCoda(Q, k)`.

Nel caso della coda, basta osservare che inserendo in ordine i nodi nella coda, la k-mediana è il **k-esimo a uscire dalla coda**.

La seconda osservazione è che **una coda è sufficiente passarla tra i parametri**: le modifiche sono visibili su Q (si opera per **side-effects**).

```
def kMedianaABR(B, k):
#REQ: k < n
    Q = newQueue()
    abrSuCoda(B, Q)
    m = kMedianaCoda(Q, k)
    return m
```

$\Theta(n+k) = \Theta(n)$

```
def abrSuCoda(B, Q):
    if B ≠ EMPTY:
        abrSuCoda(lft(B), Q)
        enqueue(Q, key[B])
        abrSuCoda(rgt(B), Q)
```

$\Theta(n)$

```
def kMedianaCoda(Q, k):
    for i=1 to k: m = dequeue(Q)
    return m
```

$\Theta(k)$

caso 2: pila

Il testo era identico, ma occorreva usare una pila. L'unica complicazione è che **dalla pila la sequenza estratta è rovesciata**.

Una soluzione era **contare il numero *n* dei nodi** dell'albero e fare $n - k$ pop, oppure **far contare i nodi** a **`abrSuPila`**, anche se questo in qualche modo **non aderiva al prototipo** proposto… ma visto che non ci sono i tipi nel nostro pseudo-Python…

La **soluzione** d'elezione era però fare una **in-order inversa** (**prima sotto-albero destro** e **poi sinistro**) inserendo le chiavi in ordine inverso.

$\Theta(n+k)$ $=\Theta(n)$

```
def kMedianaABR(B, k):
#REQ: k < n
    S = newStack()
    abrSuPila(B, S)
    m = kMedianaPila(S, k)
    return m
```

```
def kMedianaPila(Q, k):
    for i=1 to k: m = pop(Q)
    return m
```

$\Theta(k)$

$\Theta(n)$

```
def abrSuCoda(B, S):
    if B ≠ EMPTY:
        abrSuPila(rgt(B), S)
        push(S, key[B])
        abrSuPila(lft(B), S)
```

caso 3: senza strutture dati

3. ★Dovete risparmiare memoria e decidete di non allocare esplicitamente memoria di dimensione $\Theta(n)$, a parte quella (quanta?) necessaria alle chiamate ricorsive (potete quindi usare funzioni ausiliarie con più parametri "semplici" e/o più valori di ritorno, sempre "semplici").

Per risolvere questo esercizio occorreva aver ben capito la tecnica di **passare informazioni tra diverse chiamate ricorsive**: in questo caso occorre contare quanti nodi si sono visti nell'albero.

Questa informazione deve **sia andare in profondità** che **tornare al chiamante**: occorre usare sia un parametro che un valore di ritorno.

Per la cronaca, la **memoria è $\Theta(h)$**, massima profondità della ricorsione

```
def kMedianaABR(B, k):
    m, n = kMedABRAux(B, k, 1)
    return m
```

$\Theta(n)$

```
def kMedABRAux(B, k, i):
    if B==EMPTY: return 42, i
    m, i = kMedABRAux(lft(B), k, i)
    if i < 0: return m, -1
    i = i+1 # vedo un nuovo nodo
    if i==k: return key[B], -1
    return kMedABRAux(rgt(B), k, i)
```

caso 4: strutture dati aumentate

4. ★★Un'amica fidata vi dice che la k-mediana si può calcolare in tempo $\Theta(h)$ al costo di aggiungere una piccola informazione nei nodi dell'albero. Accettate la sfida, cercate di capire quale sia questa informazione, la usate nella funzione `kMedianABR`, ma a quel punto siete costretti anche a ridefinire opportunamente le funzioni `insABR` e `removeABR` per mantenere sempre coerente tale informazione dopo inserimenti e rimozioni.

Per calcolare la k-mediana in $\Theta(h)$, abbiamo la possibilità **solo di percorrere un cammino radice-foglia** nell'albero.

Era utile ricordare che k-mediana con partiziona, oltre al partizionamento, sfrutta l'informazione di sapere **quanti elementi** ci sono nella **partizione sinistra**.

Ma **un ABR è già "partizionato"** i più piccoli stanno nel sottoalbero sinistro, i più grandi in quello destro.

Quindi: basta aggiungere in ogni nodo l'informazione di quanti nodi contiene l'albero (o quanti nodi ci sono nel sottoalbero sinistro). Questa informazione **si può facilmente aggiornare** a ogni inserimento e rimozione modificandola aggiungendo/sottraendo 1 **mentre si risale dalle chiamate ricorsive** di `insAbr` e `removeABR`.