Dati e Algoritmi I (Pietracaprina)

Esercizi su Priority Queue e Heap

Problema 1 *Si definisca un* albero ternario completo di altezza h *come un albero con le seguenti proprietà: (1) ogni nodo interno ha al più 3 figli; (2) per* $0 \leq i < h$ *ci sono* 3^i *nodi al livello* i*; (3) tutti i nodi interni al livello* $h-1$ *sono alla sinistra delle eventuali foglie che stanno a quel livello e hanno tutti 3 figli tranne, eventualmente quello più a destra. (Si ricordi che dati due nodi* v *e* u *sullo stesso livello, diciamo che* v *è alla sinistra di* u *se lo precede nella visita preorder.) Sia* n *il numero di nodi dell'albero.*

a. *Dimostrare che* $h \in \Theta(\log n)$.

b. *Descrivere un'implementazione efficiente dell'albero ternario completo tramite un array.*

c. *Si supponga di memorizzare* n *entry distinte nei nodi dell'albero (una entry per nodo) con la proprietà che un nodo ha chiave minore di quelle dei suoi figli. Si ottiene così uno* heap ternario. *Descrivere e analizare un'implementazione efficiente dei metodi della Priority Queue* `insert(k,x)` *e* `removeMin()`*, sull'heap ternario.*

Soluzione.

a. Dalla definizione si ricava che:

$$\sum_{i=0}^{h-1} 3^i < n \leq \sum_{i=0}^{h} 3^i,$$

e quindi che

$$\frac{3^h - 1}{2} < n \leq \frac{3^{h+1} - 1}{2}.$$

Di conseguenza, dopo qualche semplice passaggio algebrico si ottiene che

$$h < \log_3(2n+1) \leq h+1,$$

da cui discende che

$$h = \lceil \log_3(2n+1) - 1 \rceil \in \Theta(\log n).$$

b. Sia V l'array che implementa l'albero. Associamo $V[1]$ alla radice dell'albero, e $V[3i-1], V[3i], V[3i+1]$ ai tre figli del nodo $V[i]$, per ogni $i \geq 1$. Se vi sono n nodi, l'ultimo nodo (ovvero quello più a destra nel livello h) sarà associato all'elemento $V[n]$.

c. Poniamo `last` $= n$. Per $i > 1$, si denoti con `p(i)` l'indice del padre del nodo $V[i]$, ovvero `p(i)`$=\lfloor (i+1)/3 \rfloor$. L'implementazione di `insert(k,x)` è la seguente:

```
e ⟵ (k,x)
V[last++] ⟵ entry
i ⟵ last;
while ((i>1) and (V[p(i)].getKey() > V[i].getKey())) do
    swap(V[i],V[p(i)]);
    i ⟵ p(i);
return e
```

L'implementazione di `removeMin()` è la seguente:

```
minentry ⟵ V[1];
V[1] ⟵ V[last--];
i ⟵ 1;
j ⟵ indice del figlio di V[i] con la minima chiave (se esiste);
while ((3i-1 <= last) and (V[i].getKey() > V[j].getKey())) do
    swap(V[i],V[j]);
    i ⟵ j;
    j ⟵ indice del figlio di V[i] con la minima chiave (se esiste);
return minentry
```

Entrambi i metodi hanno un costo proporzionale all'altezza h dell'albero. Dalla definizione di albero ternario completo si deduce che il numero di nodi, ovvero il numero di entry, è $n \in \Theta\left(3^h\right)$, e quindi la complessità è $\Theta(h) = \Theta(\log n)$.

□

Problema 2 *Si consideri l'implementazione vista a lezione del metodo* `insert` *di una Priority Queue P realizzata come heap su array. Dimostrarne la correttezza tramite il seguente invariante per il ciclo while:* le uniche coppie antenato-discendente che possono violare la heap-order property estesa hanno come discendente $P[i]$. (*Per "heap-order property estesa" si intende che la chiave nell'antenato deve essere $\leq$ di quella nel discendente.*)

Soluzione. Dimostriamo che l'invariante viene mantenuto nel ciclo while. All'inizio del ciclo ($i =$ `last`) è vero in quanto tutte le coppie antenato-discendente che non coinvolgono $P[i]$ soddisfano la heap-order property estesa da prima dell'inserimento. Supponiamo l'invariante vero alla fine di una data iterazione, e quindi le uniche violazioni della heap-order property estesa possono essere tra $P[i]$ e un suo antenato. Sia $e = P[i]$ ed $e' = P[\lfloor i/2 \rfloor]$. Se avviene lo swap tra e ed e' vediamo che, dopo lo swap

- non ci sono violazioni tra e (che adesso diventa $P[\lfloor i/2 \rfloor]$) e i suoi discendenti, in quanto al posto di e prima c'era e' che ha chiave maggiore.
- non ci sono violazioni tra e' (che adesso diventa $P[i]$) e i suoi discendenti, in quanto essi sono un sottoinsieme di quelli che aveva prima.
- non ci sono violazioni tra e' e i suoi antenati, in quanto, a parte il padre e, rispetto al quale e' non viola la heap-order property, gli altri antenati sono gli stessi che aveva all'inizio dell'iterazione.

Alla fine del ciclo, grazie all'invariante, le uniche possibili violazioni della heap-order property estesa possono essere tra $P[i]$ e un antenato. Se il ciclo termina perché $i = 1$, allora $P[1]$ non ha antenati e quindi non ci sono violazioni della hear-order property. Se invece il ciclo termina perché la heap-order property tra $P[i]$ e $P[\lfloor i/2 \rfloor]$ non è violata, allora dato che non può essere nemmeno violata tra $P[\lfloor i/2 \rfloor]$ e i suoi antenati (grazie all'invariante), per transitività non può essere violata nemmeno tra $P[i]$ e un antenato. Quindi non ci sono violazioni della heap-order property. In entrambi i casi si ha che P è uno heap. □

Problema 3 *Si consideri l'implementazione vista a lezione del metodo* `removeMin` *di una Priority Queue P realizzata come heap su array. Dimostrarne la correttezza tramite il seguente invariante per il ciclo while:*

- $P[j]$ figlio di $P[i]$ con chiave minima (se $P[i]$ è interno)
- Le uniche coppie antenato-discendente che possono violare la heap-order property estesa, sono coppie in cui l'antenato e' $P[i]$.

(*Per "heap-order property estesa" si intende che la chiave nell'antenato deve essere $\leq$ di quella nel discendente.*)

Soluzione. La prova ha la stessa struttura di quella fatta per il metodo `insert` nel problema precedente, e la si lascia come esercizio. □

Problema 4 *Progettare e analizzare un algoritmo per rimuovere da uno heap P con n entry la entry $P[j]$ $(1 \leq j \leq n)$, ripristinando poi le proprietà dello heap.*

Soluzione. L'idea è di rimuovere $P[j]$ mettendo al suo posto $P[last]$, in modo da mantenere la struttura di albero binario completo, e procedere poi a ripristinare le proprietà dell'heap con un up-heap bubbling o down-heap bubbling a partire dalla posizione j a seconda della relazione tra la nuova entry $P[j]$ (quella che prima era $P[last]$) e le entry del padre e dei figli di $P[j]$. In particolare, sia k la chiave della nuova entry $P[j]$, e siano k_0, k_1 e k_2 le chiavi delle entry $P[\lfloor j/2 \rfloor], P[2j]$ e $P[2j+1]$, rispettivamente, se tali entry esistono. Si hanno tre casi:

- **Caso 1:** $k_0 \leq k \leq \min\{k_1, k_2\}$. In questo caso non ci sono violazioni della heap-order property.
- **Caso 2:** $k < k_0$, e quindi $k < \min\{k_1, k_2\}$. In questo caso si esegue un up-heap bubbling a partire da $P[j]$. Siamo infatti nella condizione in cui uniche coppie discendente-antenato che violano heap-order property sono $P[j]$, $P[\ell]$ con $P[\ell]$ antenato di $P[j]$
- **Caso 3:** $k > \min\{k_1, k_2\}$, e quindi $k > k_0$. In questo caso si esegue un down-heap bubbling a partire da $P[j]$. Siamo infatti nella condizione in cui uniche coppie discendente-antenato che violano heap-order prop. sono $P[j]$, $P[\ell]$ con $P[\ell]$ discendente di $P[j]$.

Si osservi che i tre casi sono mutuamente esclusivi. L'algoritmo è il seguente:

```
Algoritmo removeEntry(P,j)
input Heap P con n entry, intero j (1 ≤ j ≤ n)
output Heap P con n-1 entry ottenuto rimuovendo P[j]

e ⟵ P[j]; P[j] ⟵ P[last--]
/* Caso 2:  up-heap bubbling a partire da P[j] */
while ((j>1) and (P[j].getKey() < P[⌊j/2⌋].getKey())) do
    swap(P[j],P[⌊j/2⌋])
    j ⟵ ⌊j/2⌋
/* Caso 3:  down-heap bubbling a partire da P[j] */
k ⟵ indice del figlio di P[j] con la minima chiave (se esiste)
while ((2j <= last) and (P[j].getKey() > P[k].getKey())) do
    swap(P[j],P[k])
    j ⟵ k
    k ⟵ indice del figlio di P[j] con la minima chiave (se esiste)
```

Si noti che solo uno dei due cicli while può essere eseguito per ogni istanza. La complessità è come quella dei metodi `insert` e `removeMin` visti a lezione, ed è quindi $\Theta(\log n)$. □

Problema 5 (Esercizio C-9.36 del testo [GTG14].) *Siano T_1 e T_2 due alberi binari (non necessariamente completi e implementati tramite struttura linkata) i cui nodi memorizzano delle entry in modo da soddisfare la heap-order property. Descrivere un algoritmo per combinare T_1 e T_2 in un unico albero binario T i cui nodi contengano tutte le entry di T_1 e T_2, mantenendo la heap-order property. La complessità dell'algoritmo deve essere $O(h_1 + h_2)$, dove h_1 e h_2 rappresentano le altezze di T_1 e T_2, rispettivamente.*

Soluzione. L'algoritmo esegue la seguente sequenza di passi: (1) si rimuove una foglia v da T_1; (2) si crea un nuovo nodo radice r contenente la entry e precedentemente contenuta in v, assegnandole T_1 e T_2 come figli sinistro e destro, rispettivamente; (3) si esegue il down-heap bubbling a partire da r. Una specifica più formale dell'algoritmo e la sua analisi sono lasciate come esercizio. □

Problema 6 (Esercizio C-9.34 del testo [GTG14].) *Progettare un algoritmo che dato uno heap T con n entry e una chiave k stampi tutte le entry in T con chiave $\leq k$. La complessità deve essere proporzionale al numero di entry stampate* ($+1$ *nel caso siano* 0).

Soluzione. Si osservi che le entry con chiave $\leq k$ formano un sottoalbero T' di T (eventualmente vuoto) con la stessa radice di T. È sufficiente quindi fare una visita di T' prestando attenzione al fatto che le foglie di T' sono foglie di T oppure nodi interni di T contenenti entry con chiave $\leq k$ ma i cui figli hanno entry con chiave $> k$. L'algoritmo è il seguente (la prima invocazione sarà `Find(T,i,k)`:

Algoritmo `Print(T,i,k)`
input `Heap T, indice i ∈[1,last], chiave k`
output `Stampa di tutte le entry con chiave ≤ k nel sottoalbero con radice T[i]`

```
if (T[i].getKey()≤k) then stampa T[i]
if ((2i≤last) AND (T[2i].getKey()≤k)) then Print(T,2i,k)
if ((2i+1≤last) AND (T[2i+1].getKey()≤k)) then Print(T,2i+1,k)
```

Se non esistono chiavi $\leq k$ in T, la complessità dell'algoritmo è chiaramente $O(1)$, altrimenti la complessità è quella di una visita in preorder di T', in cui la visita di un nodo consiste nello stampare la entry in esso contenuta. Assumendo che la stampa di una entry richieda tempo costante, la complessità della visita è $O(m+1)$. □

Problema 7 *Sia S una sequenza di n entry, $(S[1], S[2], \ldots, S[n])$, con chiavi intere distinte, e sia τ un intero in $[1, n]$. (Ad es., le entry potrebbero rappresentare film e le chiavi i loro rating.) Il seguente algoritmo trova le* Top-τ *entry di S, ovvero quelle con chiave maggiore, eseguendo un'unica scansione della sequenza.*

Algoritmo `Top(S,`τ`)`
input `Stream` S `di n entry, intero` τ `(1` $\leq \tau <<$ `n)`
output `Le` τ `entry di` S `con chiave piu' grande`

```
Q ⟵ priority queue vuota
for i ⟵ 1 to n do {
  if (i ≤ τ) then Q.insert(S[i].getKey(),S[i].getValue())
  else if (S[i].getKey() > Q.min().getKey()) then {
    Q.removeMin()
    Q.insert(S[i].getKey(),S[i].getValue())
  }
}
return Q
```

a. *Analizzare la complessità dell'algoritmo in funzione di n e di τ, assumendo di implementare Q tramite uno heap su array.*

b. *Provare la correttezza dell'algoritmo usando un opportuno invariante per il ciclo for.*

Soluzione.

a. La complessità è $O(n \log \tau)$ dato che ogni iterazione del ciclo for è dominata dalle eventuali due operazioni di insert e removeMin su Q, che contiene al più τ entry.

b. La correttezza è basata sul seguente invariante che vale alla fine di ciascuna iterazione i del for: *Q contiene le* $\min\{i, \tau\}$ *entry più grandi del prefisso* $S[1 \div i]$. È immediato vedere che l'invariante è mantenuto vero da ciascuna delle prime τ iterazioni. Supponiamo che sia vero alla fine di una certa Iterazione i, con $\tau \leq i < n$ e dimostriamo che rimane vero alla fine dell'Iterazione $i+1$. Per $\ell = i, i+1$ si denoti con Q_ℓ la coda Q alla fine della Iterazione ℓ e si definisca $A_\ell = S[1 \div \ell] - Q_\ell$. Dato che l'invariante vale alla fine dell'Iterazione i si deve avere che ogni entry in A_i ha chiave minore di quella di qualsiasi entry in Q_i. Sia $x = S[i+1]$ e $y = Q[1]$ (ovvero, y è la entry con chiave più piccola in Q). Abbiamo due casi

   - **Caso 1:** `x.getKey()` $<$ `y.getKey()`. In questo caso avremo $Q_{i+1} = Q_i$ e $A_{i+1} = A_i \cup \{x\}$.
   - **Caso 2:** `x.getKey()` $>$ `y.getKey()`. In questo caso avremo $Q_{i+1} = (Q_i - \{y\}) \cup \{x\}$ e $A_{i+1} = A_i \cup \{y\}$.

   In entrambi i casi vale che per ogni $z \in A_{i+1}$ e $w \in Q_{i+1}$ la chiave di z è minore di quella di w (la prova è lasciata come esercizio), e quindi l'invariante continua a valere.

□

Problema 8 *Sia P uno heap contenente n entry con chiavi distinte. Dato un intero m, con $1 \leq m \leq n$, il seguente algoritmo trova le m entry di P con chiave più piccola restituendole in una sequenza S in ordine crescente di chiave. Ovvero, alla fine $S[i]$ deve contenere la entry (e_i) con la i-esima chiave più piccola tra quelle di P, per $1 \leq i \leq m$. L'algoritmo fa uso di uno priority queue Q di appoggio, implementata tramite heap su array, le cui entry corrispondono a entry di P ma dove il valore è il loro indice in P.*

Algoritmo `Find(P,m)`
input `Heap P con n entry distinte, intero m (1 ≤ m ≤ n)`
output `Sequenza ordinata S con le m entry con chiave piu' piccola`

```
Q ⟵ priority queue vuota; S ⟵ sequenza vuota;
S[1] ⟵ P[1];
if (2 ≤ n) then Q.insert(P[2].getKey(),2);
if (3 ≤ n) then Q.insert(P[3].getKey(),3);
for i ⟵ 2 to m do
  (k,j) ⟵ Q.removeMin(); // k=chiave di P[j]
  S[i] ⟵ P[j];
  if (2j ≤ n) then Q.insert(P[2j].getKey(),2j);
  if (2j+1 ≤ n) then Q.insert(P[2j+1].getKey(),2j+1);
return S
```

a. *Applicare l'algoritmo con $n = 10$ e $m = 4$ al seguente heap P*

	(3,B)	(7,R)	(9,D)	(10,H)	(8,V)	(12,L)	(15,C)	(11,A)	(16,M)	(18,Z)

in cui le chiavi sono interi e i valori sono lettere, facendo vedere lo stato di Q edi S all'inzio del ciclo e alla fine di ogni iterazione.

b. *Per un'istanza generica, cosa contengono Q ed S alla fine di ogni iterazione?*

c. *Analizzare la complessità in tempo dell'algoritmo.*

Soluzione.

a. Nell'esempio dato, Q ed S si evolvono come segue.

- All'inizio del ciclo:

$$\begin{aligned} S &= [(3, B)] \\ Q &= [(7, 2)(9, 3)] \end{aligned}$$

- Alla fine della iterazione 2:

$$\begin{aligned} S &= [(3, B)(7, R)] \\ Q &= [(8, 5)(10, 4)(9, 3)] \end{aligned}$$

- Alla fine della iterazione 3:

$$\begin{aligned} S &= [(3, B)(7, R)(8, V)] \\ Q &= [(9, 3)(10, 4)(18, 10)] \end{aligned}$$

- Alla fine della iterazione 4:

$$\begin{aligned} S &= [(3,B)(7,R)(8,V)(9,D)] \\ Q &= [(10,4)(15,7)(12,6)(18,10)] \end{aligned}$$

b. Il seguente invariante vale alla fine di ogni iterazione $i \geq 1$ del ciclo for (la fine dell'interazione 1 è l'inizio del ciclo).

- $S[j] = e_j$, per $1 \leq j \leq i$.
- Le entry in Q sono tutte quelle entry e_ℓ con $\ell > i$ che in P sono figlie di entry che sono attualmente S (ovvero entry e_j con $j \leq i$).

c. Per quanto riguarda la complessità si osservi che fuori dal ciclo `for` vengono eseguite $\Theta(1)$ operazioni, e che in ciascuna iterazione del ciclo vengono eseguite al più tre operazioni su Q (una `removeMin` e due `insert`) e $\Theta(1)$ altre operazioni. Poichè Q contiene inizalmente due entry e in ogni iterazione la sua taglia aumenta al più di uno, Q non contiene mai più di $m+1$ entry, quindi il costo delle `removeMin` e `insert` eseguite in ciascuna iterazione è $O(\log m)$. Se ne deriva che la complessità dell'algoritmo è $O(m \log m)$. È importante notare che la complessità non dipende da n in quanto nessuna operazione costosa viene fatta su P.

□

Problema 9 *Sia $S = S[1], S[2], \ldots, S[n]$ una sequenza contenente n chiavi distinte e sia m un indice intero tale che $1 \leq m \leq n/\log_2 n$. Progettare un algoritmo che in tempo $O(n)$ determini la m-esima chiave più piccola di S. (Analizzare la complessità dell'algoritmo per far vedere che è $O(n)$.)*

Soluzione. L'idea è quella di costruire prima uno heap contenente le n chiavi (viste come entry), e poi estrarre per m volte la chiave minima usando la stessa strategia di `removeMin()`. L'ultima chiave estratta è quella cercata e la si restituisce in output. Per risparmiare spazio è possibile utilizzare S sia per contenere lo heap che per memorizzare le chiavi di volta in volta estratte che vengono sistemate a partire dal fondo di S. Lo pseudocodice è il seguente:

Algoritmo `Find(S,m)`
input `Insieme S con n chiavi distinte, indice m tale che` $1 \leq$ `m` $\leq$ `n/`$\log_2$ `n`
output `m-esima chiave piu' piccolo in S`

```
Trasforma S in uno heap usando la costruzione bottom-up (solo chiavi);
for j ⟵ 1 to m do {
  swap(S[1],S[n-j+1]);
  last ← n-j;
  i ← 1;
  if (2i ≤ last) then k ← figlio di S[i] con chiave minima;
  while ((2i ≤ last) and (S[i]>S[k])) do {
    swap(S[i],S[k]);
    i ← k;
```

```
    if (2i ≤ last) then k ← figlio di S[i] con chiave minima;
  }
}
return S[n-j+1]
```

La complessità risulta essere $O(n)$ in quanto la costruzione bottom-up dello heap richiede $O(n)$ operazioni, e ciascuna delle $m \leq n/\log_2 n$ iterazioni del for consiste essenzialmente in uno swap e un down-heap bubbling in uno heap con $< n$ chiavi, e richiede quindi $O(\log n)$ operazioni. □