Dati e Algoritmi (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. Sia* n *il numero di nodi dell'albero.*

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

b. *Descrivere un mapping efficiente dell'albero ternario completo su un array* V*, facendo vedere dove vengono mappati i figli di un nodo interno* $V[i]$*, e dove viene mappato il padre di un nodo non radice* $V[i]$.

c. *Sia* V *un albero ternario completo, su array, i cui nodi memorizzano* n *entry distinte, una entry per nodo, con la proprietà che la entry in un nodo ha chiave minore di quelle nei suoi figli* (heap ternario)*. Descrivere e analizare un'implementazione efficiente del metodo* `insert(k,x)` *della Priority Queue.*

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. Associamo $V[0]$ alla radice dell'albero e $V[3i+1], V[3i+2], V[3i+3]$ ai tre figli del nodo $V[i]$, per ogni $i \geq 0$. Di conseguenza, il padre di $V[i]$ sarà $V[\lfloor (i-1)/3 \rfloor]$. Si osservi che se vi sono n nodi, l'ultimo nodo (ovvero quello più a destra nel livello h) sarà $V[n-1]$.

c. Poniamo `last` $= n-1$. Per $i > 0$, 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>0) and (V[p(i)].getKey() > V[i].getKey())) do
    swap(V[i],V[p(i)]);
    i ⟵ p(i);
return e
```

La complessità del metodo è proporzionale all'altezza h, ed è quindi $\Theta(\log n)$, in base a quanto provato nel primo punto.

□

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 violazioni della heap-order property estesa possono essere tra $P[i]$ e un suo antenato. (*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. Lo stato iniziale da cui si parte è quello in cui in P, a valle dell'inserimento, la heap-order property estesa vale per tutte le coppie tranne, eventualmente quelle che coinvolgono $P[\texttt{last}]$ che, essendo una foglia, può essere solo discendente nella coppia. Quindi all'inizio del ciclo ($i = \texttt{last}$) l'invariante è vero. 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 $e' = P[\lfloor (i-1)/2 \rfloor]$. Se avviene lo swap tra e ed e' vediamo che, dopo lo swap

- non ci sono violazioni tra e (che adesso diventa $P[\lfloor (i-1)/2 \rfloor]$) e i suoi discendenti, in quanto tutti i discendenti di e erano prima discendenti di 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 $P[\lfloor (i-1)/2 \rfloor]$ non è violata, allora dato che non può essere nemmeno violata tra $P[\lfloor (i-1)/2 \rfloor]$ 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, che è la proprietà finale desiderata. □

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 violazioni della heap-order property estesa possono essere tra $P[i]$ e un suo discendente.

(*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 $P[\ell]$ $(0 \leq \ell \leq n-1)$, ripristinando poi le proprietà dello heap.*

Soluzione. L'idea è di rimuovere $P[\ell]$ mettendo al suo posto $P[\text{last}]$, 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 ℓ a seconda della relazione tra la nuova entry $P[\ell]$ (quella che prima era $P[\text{last}]$) e le entry del padre e dei figli di $P[\ell]$. In particolare, sia k la chiave della nuova entry $P[\ell]$, e siano k_0, k_1 e k_2 le chiavi delle entry $P[\lfloor(\ell-1)/2\rfloor], P[2\ell+1]$ e $P[2\ell+2]$, 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 $P[\ell]$. Siamo infatti nella condizione in cui le uniche coppie antenato-discendente che violano heap-order property estesa hanno come discendente $P[\ell]$.
- **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 $P[\ell]$. Siamo infatti nella condizione in cui le uniche coppie antenato-discendente che violano heap-order property estesa hanno come antenato $P[\ell]$.

Si osservi che i tre casi sono mutuamente esclusivi. Sia `indexMinChild(P,i)` il metodo che restituisce l'indice del figlio di $P[i]$ con chiave minima ($2i+1$ o $2i+2$), se P[i] è un nodo interno, e restituisce `null`, se $P[i]$ è foglia. Lo pseudocodice è il seguente:

Algoritmo `removeEntry(P,`ℓ`)`
input `Heap P con n entry, intero` ℓ `(0` $\leq$ ℓ $\leq$ `n-1)`
output `Heap P con n-1 entry ottenuto rimuovendo P[`ℓ`]`

```
P[ℓ] ⟵ P[last--]; i ⟵ ℓ;
/* Caso 2:  up-heap bubbling a partire da P[j] */
while ((i≥1) and (P[i].getKey() < P[⌊(i-1)/2⌋].getKey())) do {
    swap(P[i],P[⌊(i-1)/2⌋])
    i ⟵ ⌊(i-1)/2⌋
}
/* Caso 3:  down-heap bubbling a partire da P[j] */
j ⟵ indexMinChild(P,i);
while ((j != null) and (P[i].getKey() > P[j].getKey())) do {
    swap(P[i],P[j])
    i ⟵ j
    j ⟵ indexMinChild(P,i);
}
```

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-9.34 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* ($\Theta(1)$ *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 `Print(T,0,k)`:

Algoritmo `Print(T,i,k)`
input `Heap T, indice i ∈[0,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] // Visita di T[i]
  if ((2i+1≤last) AND (T[2i+1].getKey()≤k)) then Print(T,2i+1,k)
  if ((2i+2≤last) AND (T[2i+2].getKey()≤k)) then Print(T,2i+2,k)
}
```

Se non esistono chiavi $\leq k$ in T, la complessità dell'algoritmo è chiramente $O(1)$, 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 $\Theta(m+1)$. □

Problema 6 (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. Un'idea semplice per combinare i due alberi è quella di creare un nuovo nodo radice r in cui si memorizza una entry rimuovendola da una foglia qualsiasi di uno dei due alberi, collegare i due alberi come figli di r, e poi eseguire il down-heap bubbling da r. I passi sono i seguenti:

a. Crea un nuovo nodo r come radice di T.

b. Trova una foglia v in T_1 (ad es., scendendo dalla radice lungo un percorso qualsiasi sino a una foglia v).

c. Sia e la entry in v: metti la entry e in r e rimuovi v da T_1

d. Collega le radici di T_1 e T_2 come figli di r.

e. Esegui un adattamento del down-heap bubbling in T a partire da r.

Una specifica più dettagliata dell'algoritmo è lasciata come esercizio. Per quanto riguarda la complessità, il Passo (b) richiede $O(h_1)$ operazioni, il Passo (e) richiede $O(\max\{h_1, h_2\}) = O(h_1 + h_2)$ operazioni, e tutti gli altri passi richiedono $O(1)$ operazioni, quindi la complessità finale è $O(h_1 + h_2)$. □

Problema 7 *Sia $S = S[0], S[1], \ldots, S[n-1]$ una sequenza di n entry con chiavi intere distinte (ad es., profili IG con i rispettivi numeri di follower), e sia τ un intero in $[1, n]$. Si vuole progettare un algoritmo che trovi le τ entry in S con chiave maggiore* (Top-τ entry) *basato sul seguente schema.*

Algoritmo `Top`(S,τ)
Input: *Sequenza S di n entry, intero $\tau \leq n$*
Output: *τ entry di S con chiave più grande*

```
Q ⟵ priority queue vuota;
for i ← 0 to n-1 do
    if (i < τ) then Q.insert(S[i].getKey(),S[i].getValue());
    else
        
return Q
```

a. *Completare l'algoritmo in modo che sia corretto e che Q non contegna mai più di τ entry.*

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

c. *Analizzare la complessità assumendo che Q sia uno heap su array.*

Soluzione.

a. L'algoritmo completo è il seguente:

```
Q ⟵ priority queue vuota;
for i ← 0 to n-1 do
    if (i < τ) then Q.insert(S[i].getKey(),S[i].getValue());
    else
        if (Q.min().getKey() < S[i].getKey()) then
            Q.removeMin();
            Q.insert(S[i].getKey(),S[i].getValue())
return Q
```

b. La correttezza è basata sul seguente invariante che vale alla fine di ciascuna iterazione i del for: *Q contiene le* $\min\{i+1, \tau\}$ *entry in $S[0 \div i]$ con chiave maggiore.* Lo stato iniziale è quello in cui Q è vuota. È immediato vedere l'invariante vale alla fine dell'Iterazione i, per $-1 \leq i \leq \tau - 1$, dove la fine dell'iterazione $i-1$ rappresenta l'inizio del ciclo. Supponiamo che sia vero alla fine di una certa Iterazione i, con $\tau - 1 \leq i < n$ e dimostriamo che rimane vero alla fine dell'Iterazione $i+1$. Dato che l'invariante vale alla fine dell'Iterazione i si deve avere che ogni entry in $S[0 \div i] - Q$ ha chiave minore di quella di qualsiasi entry in Q. Sia $x = S[i+1]$ e $y = Q[0]$ (ovvero, y è la entry con chiave più piccola in Q). È facile vedere che se la chiave di x è più piccola di quella di y allora le entry in Q sono anche le τ entry con chiave più grande del prefisso $S[0 \div i+1]$, mentre se la chiave di x è più grande di quella di y, allora le τ entry con chiave più

grande del prefisso $S[0 \div i+1]$ sono costituite da x e da tutte quelle di Q, tranne y. Ne consegue che le operazioni eseguite dall'algoritmo nell'Iterazione $i+1$ mantengono vero l'invariante alla fine di tale iterazione. Alla fine dell'ultima iterazione, l'invariante implica immediatamente la proprietà finale desiderata, ovvero che Q contiene le τ chiavi più grandi di S.

c. Assumendo di implementare Q tramite heap su array, 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.

□

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 un array S in ordine crescente di chiave. Ovvero, alla fine $S[i-1]$ 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 ⟵ array vuoto;
S[0] ⟵ P[0];
if (1 ≤ n) then Q.insert(P[1].getKey(),1);
if (2 ≤ n) then Q.insert(P[2].getKey(),2);
for i ⟵ 2 to m do {
  (k,j) ⟵ Q.removeMin(); // k=chiave di P[j]
  S[i] ⟵ P[j];
  if (2j+1 ≤ n) then Q.insert(P[2j+1].getKey(),2j+1);
  if (2j+2 ≤ n) then Q.insert(P[2j+2].getKey(),2j+2);
}
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:

$$\begin{aligned} S &= [(3,B)] \\ Q &= [(7,1)(9,2)] \end{aligned}$$

- Alla fine della iterazione 2:

$$\begin{aligned} S &= [(3,B)(7,R)] \\ Q &= [(8,4)(10,3)(9,2)] \end{aligned}$$

- Alla fine della iterazione 3:

$$\begin{aligned} S &= [(3,B)(7,R)(8,V)] \\ Q &= [(9,2)(10,3)(18,9)] \end{aligned}$$

- Alla fine della iterazione 4:

$$\begin{aligned} S &= [(3,B)(7,R)(8,V)(9,D)] \\ Q &= [(10,3)(15,6)(12,5)(18,9)] \end{aligned}$$

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).

- $S[j-1] = e_j$, 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 $S = S[0], S[1], \ldots, S[n-1]$ 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 con valore nullo), 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 non perdere le chiavi di S 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. Sia `indexMinChild(S,i,last)` un metodo che se $2i+1 >$ `last` restituisce `null`, altrimenti restituisce l'indice, tra $2i+1$ e $2i+2$, che sia $\leq$ `last` e contenga la chiave minima (last sarà l'indice dell'ultima chiave del prefisso di S le cui chiavi soddisfano la heap-order property). Lo pseudocodice dell'algoritmo è il seguente:

Algoritmo `Find(S,m)`
input `Sequenza S con n chiavi distinte, indice m` $\in [1, n/\log_2 n]$
output `m-esima chiave piu' piccola in S`

```
last ← n-1;
for j ⟵ ⌊ (n-2)/2 ⌋ downto 0 do {
  i ← j;
  k ← indexMinChild(S,i);
  while ((k != null) and (S[i]>S[k])) do {
    swap(S[i],S[k]);
    i ← k;
    k ← indexMinChild(S,i);
  }
}
for j ⟵ 1 to m-1 do {
  swap(S[n-j],S[0]);
  last ← n-j-1;
  i ← 0;
  k ← indexMinChild(S,i);
  while ((k != null) and (S[i]>S[k])) do {
    swap(S[i],S[k]);
    i ← k;
    k ← indexMinChild(S,i);
  }
}
return S[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. □