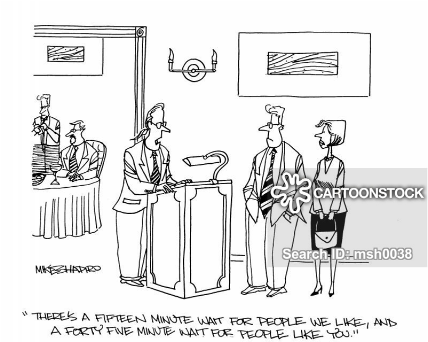
Algoritmi e strutture dati

Heaps, *HeapSort*, e code di priorità



Menú di questa lezione

In questa lezione vedremo, prima, la struttura dati heap, in due versioni, e le operazioni ad essa associata. Poi studieremo due applicazioni: *HeapSort* e code di priorità.

Stotice

Introduciamo il concetto di min e di max-heap. Una heap è una struttura dati astratta, parzialmente basata sull'ordinamento, e necessariamente compatta. Sará dunque basata su array. La caratteristica principale è quella che una heap mantiene le chiavi semi-ordinate, Useremo le heap come base per le code di prioritá (che a loro volta sono strutture dati astratte), ma anche come base per un nuovo algoritmo di ordinamento che risolve il maggior problema di *MergeSort* (quello di non essere in place) ed il maggior problema di *QuickSort* (quello di avere un tempo quadratico nel caso peggiore). E' importante non confondere il nome generico di questa struttura dati, min oppure max-heap, con il nome generico della memoria principale nel modello classico: heap, opposto a stack.

Il concetto di heap, inizialmente legato a quello di albero binario, fu introdotto assieme a *HeapSort* (che vedremo) da J.W.J. Williams, nel 1964.

Heap binarie su array

Una **(min/max) heap** è un array H che puó essere visto come un albero binario **quasi completo**, cioè tale da avere tutti i livelli pieni, meno, eventualmente, l'ultimo. I nodi dell'albero corrispondono agli elementi dell'array. L'elemento $H[1]$ dell'array è la **radice** dell'albero e, normalmente, si tende a differenziare i valori $H.length$ (lunghezza dell'array che **contiene** una heap - H è un array e assumiamo sempre presente il campo $H.length$) e $H.heapsize$ (numero di elementi della heap contenuta in H). Tipicamente si ha che $0 \leq H.heapsize \leq H.length$. Dobbiamo stare attenti a non confondere il fatto che una heap sia un array con il fatto che **per convenienza** si puó visualizzare come albero. La nozione di albero e di grafo la riprenderemo formalmente piú avanti; dal punto di vista algebrico non ci sono differenze, ma dal punto di vista delle strutture dati le heap **non** sono alberi.

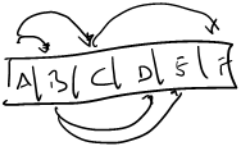
Heap binarie su array

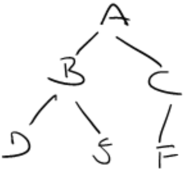
La corretta implementazione di una heap prevede che i **figli** di un nodo nella posizione i siano precisamente gli elementi nelle posizioni $2 \cdot i$ e $2 \cdot i + 1$ (sinistro e destro rispettivamente). Per conseguenza, il **padre** di un nodo i è identificato dall'indice $\lfloor \frac{i}{2} \rfloor$.

proc ***Parent*** (i)
$\{$**return** $\lfloor \frac{i}{2} \rfloor$

proc ***Left*** (i)
$\{$**return** $2 \cdot i$

proc ***Right*** (i)
$\{$**return** $2 \cdot i + 1$





Heap binarie su array

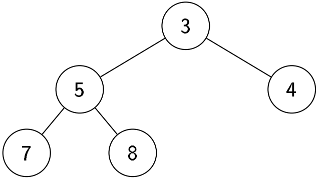
Immaginiamo una variabile H, che è un array di interi con un campo aggiuntivo $H.heapsize$. Distinguiamo tra due tipi di heap: **max-heap** e **min-heap**. Entrambe soddisfano una proprietá: nel primo caso, abbiamo che per ogni i, $H[Parent(i)] \geq H[i]$, e nel secondo caso, per ogni i, $H[Parent(i)] \leq H[i]$. Conseguentemente, il massimo elemento di una max-heap si trova alla radice, mentre nel caso di una min-heap è il minimo elemento a trovarsi alla radice. Vista come un albero binario, l'**altezza** di una heap è la lunghezza del massimo cammino (numero di archi) dalla radice ad una foglia.

Heap binarie su array

Ecco un esempio di min-heap:

minimo elemento

altezza 2

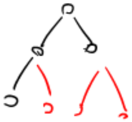


H :

3	5	4	7	8
1	2	3	4	5

[illegible]

	min	max
$h=0$	1	1
$h=1$	2	3
$h=2$	4	7
⋮		
h	2^h	$2^{h+1}-1$



Heap binarie su array: calcolo dell'altezza

Se una heap (indipendentemente se min- o max-) ha altezza h, quali sono il minimo ed il massimo numero di elementi che puó contenere? É facile calcolare questo numero semplicemente guardando i casi estremi. Una heap di altezza zero, puó contenere esattamente un elemento. Una di altezza 1, puó contenerne 2 (come minimo) o 3 (al massimo). Una di altezza 2, ha come minimo 4 elementi, e, al massimo, 7. In generale: una heap di altezza h contiene **al minimo** 2^h elementi, ed **al massimo** $2^{h+1} - 1$. Questa proprietà (che potremmo mostrare formalmente per induzione) dipende esclusivamente dal fatto che la heap è (come) un albero binario quasi completo. Da qui otteniamo che

$$2^h \leq n \leq 2^{h+1} - 1 \Rightarrow 2^h \leq n < 2^{h+1} \Rightarrow h \leq log(n) < h + 1.$$

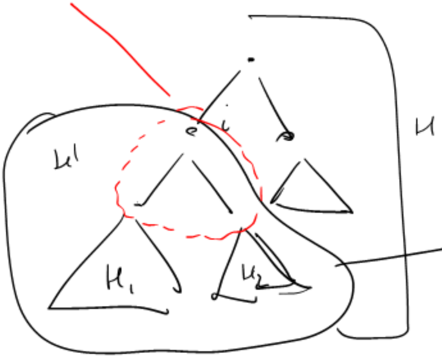
Quindi, dalla prima si ottiene che $h \leq log(n))$ cioè $h = O(log(n))$, e dalla seconda che $h > log(n) - 1$ cioè $h = \Omega(log(n))$. Pertanto $h = \Theta(log(n))$.

Heap binarie su array: *BuildMinHeap* e *MinHeapify*

Affrontiamo quindi il problema: data una (non)heap H di numeri interi non negativi (cioè un array), trasformarlo in una min-heap. A questo fine, risolviamo prima un problema piú semplice: dato un array H, ed un indice i su di esso tale che $H[Left(i)]$ e $H[Right(i)]$ sono *giá* delle min-heap, trasformare H in un array tale che anche $H[i]$ è una min-heap. Procediamo in maniera ricorsiva: sistemiamo il potenziale errore localmente a $i, Left(i), Right(i)$, e poi correggiamo ricorsivamente gli errori che vengono generati dalla sistemazione a livelli piú bassi. Vediamo una procedura che si chiama *MinHeapify* che fa quanto detto. Nel proseguio peró (e anche negli esercizi) potremmo fare riferimento a max-heap, invece che min-heap, e alla procedura simmetrica *MaxHeapify*.

Qui potrebbe non essere rispettata la condizione min-heap

Situazione ipotizzata prima di chiamare min-heapify(H, i)



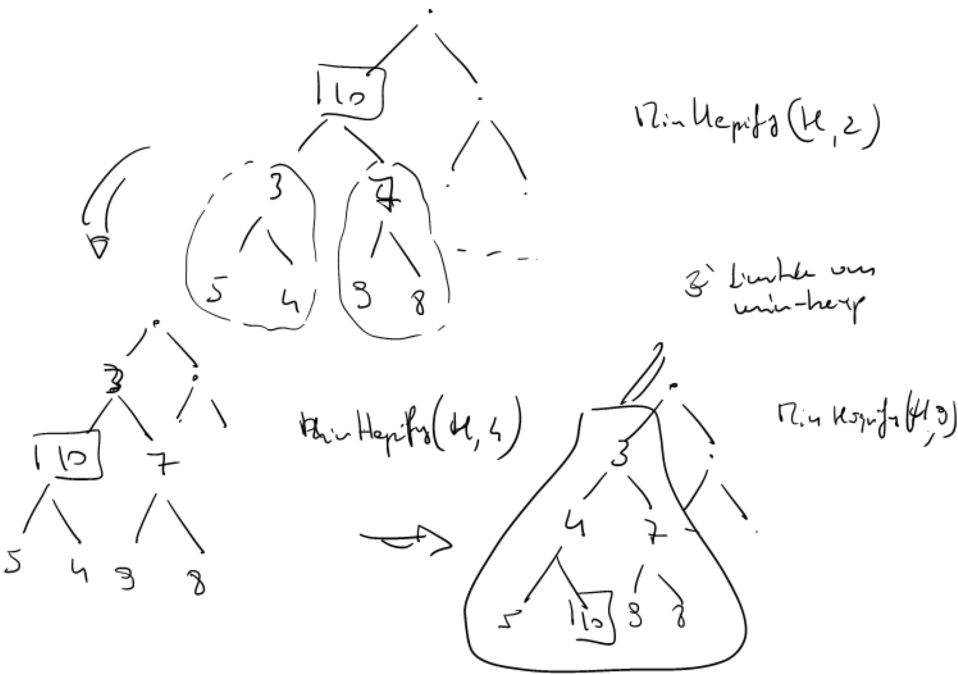
Non è ancora una min-heap

$\underbrace{H_1 \text{ e } H_2 \text{ sono già delle min-heap}}$

proc ***MinHeapify*** (H, i)

- $l = Left(i)$
- $r = Right(i)$
- $smallest = i$
- **if** $((l \leq H.heapsize)$ **and** $(H[l] < H[i]))$
  - **then** $smallest = l$
- **if** $((r \leq H.heapsize)$ **and** $(H[r] < H[smallest]))$
  - **then** $smallest = r$
- **if** $smallest \neq i$
  - **then**
    - $SwapValue(H, i, smallest)$
    - $MinHeapify(H, smallest)$

A questo punto 'smallest' è l'indice dell'elemento minore tra i, left(i), right(i)


MinHeapify(H,2)
3 Wurzel von min-heap
MinHeapify(H,4)
MinHeapify(H,9)
10
3
7
5
4
9
8

Correttezza e complessità di *MinHeapify*

Come per tutti gli algoritmi non immediati, ci poniamo il problema di mostrarne la correttezza. Per la **terminazione**, osserviamo che la procedura termina in due casi: o perchè l'indice i non cambia durante una esecuzione (nel qual caso non si effettuano chiamate ricorsive), oppure perchè è diventato piú grande della dimensione della heap; poichè se cambia cresce sempre, una delle due condizioni deve per forza essere verificata prima o poi. Per quanto riguarda la **correttezza**, osserviamo che *MinHeapify* è costruita in maniera ricorsiva. Quindi, dobbiamo trovare una **invariante** ricorsiva: dopo ogni chiamata a *MinHeapify* su un nodo di altezza h tale che entrambi i figli sono radici di min-heap prima della chiamata, quel nodo è la radice di una min-heap.

Correttezza e complessità di *MinHeapify*

Dimostriamo che l'invariante vale.

- Supponiamo, come **caso base**, *MinHeapify* venga chiamata su un nodo ad altezza $h = 0$. Le ipotesi sono rispettate, perchè il nodo è una foglia; inoltre la procedura non ha alcun effetto, ma, allo stesso tempo, un nodo senza figli è già una min-heap. Quindi, nel caso base, l'invariante ricorsiva è rispettata.
- Per il **caso induttivo**, consideriamo un nodo in posizione i ad altezza $h > 0$. Sappiamo che entrambi i suoi figli, in posizioni $2 \cdot i$ e $2 \cdot i + 1$, se esistono, sono radici di min-heap per ipotesi. La procedura sceglie il minimo tra $H[i], H[2 \cdot i]$, e $H[2 \cdot i + 1]$ e lo mette in posizione $H[i]$. Poi effettua una chiamata ricorsiva sull'indice $2 \cdot i$ oppure $2 \cdot i + 1$, che sono, entrambi, ad altezza inferiore a h. Per ipotesi induttiva, dopo la chiamata ricorsiva, quel nodo sarà radice di una min-heap, e, per ipotesi, il fratello è ancora radice di una min-heap. Poichè $H[i]$ è il minimo tra $H[i], H[2 \cdot i]$, e $H[2 \cdot i + 1]$, allora anche il nodo i è radice di una min-heap, come volevamo.

Caso peggiore nel calcolo della complessità di MinHeapify

Quando si verifica in caso di sbilanciamento uno o una volta è limitato dalla struttura dell'albero quasi completo

Caso peggiore = ① Sbilanciamento di 1 livello

② Sx pieno al massimo
Dx pieno al ~~minimo~~ massimo

$Sx:\quad 2^{h}-1$

$Dx:\quad 2^{h-1}-1$

$TOT:\quad 2^{h}+2^{h-1}-2+1$

$$\frac{Sx}{TOT}=\frac{2^{h}-1}{2^{h}+2^{h-1}-1}$$

NON SO CALCOLARE ESATTAMENTE

$\Rightarrow$ PASSO AL LIMITE

$$\lim_{h\to\infty}\frac{2^{h}-1}{2^{h}+2^{h-1}-1}=\frac{\cancel{2^{h-1}}\left(2-\overbrace{\frac{1}{2^{h-1}}}^{\to 0}\right)}{\cancel{2^{h-1}}\left(2+1-\underbrace{\frac{1}{2^{h-1}}}_{\to 0}\right)}=\frac{2}{3}$$

Per calcolare la **complessità** di *MinHeapify*, dobbiamo costruire una ricorrenza. Incorriamo in due problemi: primo, dobbiamo capire qual è il caso peggiore, e, secondo, dobbiamo renderci conto che, da un lato vorremmo che come sempre la complessità fosse in funzione della quantità di elementi nella struttura dati, e dall'altro, invece, la complessità di *MinHeapify* dipende dall'altezza dell'elemento su cui è richiamato. Per ovviare a entrambe queste difficoltà, facciamo le seguenti osservazioni. In primo luogo, il caso peggiore occorre quando la heap sulla quale la procedura è richiamata tende ad essere sbilanciata, forzando piú chiamate ricorsive. Possiamo mostrare che il peggior sbilanciamento possibile è $\frac{2}{3}$. Inoltre, poichè vogliamo un risultato che possiamo usare in ogni situazione, utilizziamo una ricorrenza leggermente piú debole, cioè:

$$T(n) \leq T(\frac{2}{3}n) + \Theta(1).$$

Perché sto tralasciando x semplicità

$a = 1$
$b = \frac{3}{2}$

Risolvendo la ricorrenza classica associata a quella precedente si ottiene che $T(n) = \Theta(log(n))$ (Master Theorem, caso 2). Pertanto, affermiamo che nel caso peggiore *MinHeapify* costa $\boxed{O(log(n))}$. Questo, ripetiamo, è un'approssimazione dovuta al fatto che dovremmo calcolare la complessità in base all'altezza del nodo su cui la procedura viene chiamata; una forma alternativa di scrivere questo risultato è dire che la complessità è $O(h)$, dove h è l'altezza della heap (e di nuovo, stiamo approssimando perchè non teniamo conto dell'altezza reale del nodo).

Ci poniamo adesso il problema originale: dato un array H di interi, convertirlo in una min-heap.

proc ***BuildMinHeap*** (H)
$\begin{cases} H.heapsize = H.length \\ \textbf{for } (i = \lfloor \frac{H.length}{2} \rfloor \textbf{ downto } 1)\ MinHeapify(H, i) \end{cases}$

Esempio : $H.length = 11$

$\left\lfloor \frac{H.length}{2} \right\rfloor$

Correttezza e complessitá di *BuildMinHeap*

Adesso mostriamo la correttezza di *BuildMinHeap*. La **terminazione** della procedura è ovvia. Per la **correttezza**, l'**invariante** che usiamo è: all'inizio di ogni iterazione del ciclo **for**, ogni elemento $H[i+1], H[i+2], \ldots$ è la radice di una min-heap, e all'uscita dall'iterazione, anche $H[i]$ lo è.

Dimostriamo:

- Nel **caso base** $i = \lfloor \frac{A.length}{2} \rfloor$: in questo caso ogni elemento del tipo $H[i+k]$ con $k > 0$ è una foglia, e pertanto la radice di una min-heap triviale di un solo elemento.
- Nel **caso induttivo**, è sufficente riferirsi alla correttezza di *MinHeapify*. Si noti che questa proprietà, riferita all'uscita dal ciclo, dice: $H[1]$ é una min-heap, che è ciò che volevamo.

Correttezza e complessitá di *BuildMinHeap*

Un calcolo della **complessità** approssimativo ci porterebbe alla seguente conclusione: ogni chiamata di *MinHeapify* costa $O(log(n))$ nel caso peggiore, e si chiama $\Theta(n)$ volte, pertanto il costo totale è $O(n \cdot log(n))$. Peró, in questo caso possiamo dare un limite piú stretto grazie ad una analisi piú dettagliata, dovuto all'approssimazione utilizzata precedentemente nel calcolare la complessità di *MinHeapify*. Infatti, come ricordamo, il costo di *MinHeapify* nel caso peggiore puó essere espresso come $O(h)$; scegliamo adesso di supporre che h sia l'altezza della **nodo** su cui viene chiamato, e non della heap nella sua interezza. Una semplice osservazione ci dice che se in un albero binario quasi completo ci sono n elementi, allora al massimo $\lceil \frac{n}{2^{h+1}} \rceil$ di loro si trovano ad altezza h.

SI OSSERVA CHE MOLTI ELEMENTI SONO MOLTO "BASSI" E QUINDI NON POSSO APPROSSIMARE MINHEAPIFY CON $O(\lg(n))$. USO COME APPROSSIMAZIONE $O(h)$ DOVE h È L'ALTEZZA DEL NODO SU CUI VIENE CHIAMATO

In generale, ad altezza h ci sono al massimo $\left\lceil \frac{n}{2^{h+1}} \right\rceil$

$h = 1 \qquad \frac{n}{4}$

$h = 0 \qquad \frac{n}{2}$

Correttezza e complessitá di *BuildMinHeap*

Quindi il costo totale, nel caso peggiore, cioè quando *MinHeapify* deve sempre arrivare alle foglie, si può limitare con:

$$\boxed{\Sigma_{h=0}^{log(n)}(\lceil\frac{n}{2^{h+1}}\rceil \cdot O(h))} = O(n \cdot \Sigma_{h=0}^{log(n)}\frac{h}{2^h})$$

Infatti l'altezza va da 0 a $log(n)$, e per una fissata altezza h, ci sono $\lceil\frac{n}{2^{h+1}}\rceil$ nodi. Per ogni nodo ad altezza h, chiamare *MaxHeapify* costa $O(h)$, e quindi $\Sigma_{h=0}^{log(n)}(\lceil\frac{n}{2^{h+1}}\rceil \cdot O(h))$. Ma n non dipende da h (e quindi moltiplica la sommatoria) e $\frac{1}{2^{h+1}} \cdot h$ si può maggiorare con $\frac{h}{2^h}$, e quindi è $O(n \cdot \Sigma_{h=0}^{log(n)}\frac{h}{2^h})$.

Correttezza e complessitá di *BuildMinHeap*

$$O(n \cdot \Sigma_{h=0}^{log(n)} \frac{h}{2^h}) =$$
$$O(n \cdot \Sigma_{h=0}^{log(n)} h \cdot (\tfrac{1}{2})^h) =$$
$$O(n \cdot \Sigma_{h=0}^{\infty} h \cdot (\tfrac{1}{2})^h) =$$
$$O(n)$$

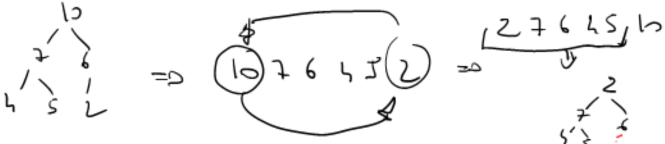
Serie convergente

La sommatoria $\Sigma_{h=0}^{log(n)} h \cdot (\frac{1}{2})^h$ è sostituita dalla serie infinita $\Sigma_{h=0}^{\infty} h \cdot (\frac{1}{2})^h$ perchè quest'ultima converge ad una costante e, poi scompare nella notazione Θ. Il fatto che $\Sigma_{h=0}^{\infty} h \cdot (\frac{1}{2})^h$ converga a una costante si ha perchè

$$lim_{h\to\infty} \frac{(h+1)\cdot(\frac{1}{2})^{h+1}}{h\cdot(\frac{1}{2})^h} = \frac{1}{2},$$

e, per il teorema del rapporto, questa è condizione sufficiente per la convergenza. Il caso migliore, cioè quello in cui l'array è già una min-heap prima di chiamare *BuildMinHeap*, ha comunque costo $\Theta(n)$, perchè la procedura è governata da un ciclo **for**.

$$\lim_{n\to\infty} \frac{(n+1)\left(\frac{1}{2}\right)^{n+1}}{n\left(\frac{1}{2}\right)^{n}} = \lim_{n\to\infty} \frac{\cancel{\left(\frac{1}{2}\right)^{n}}\cdot\frac{1}{2}\cdot(n+1)}{\cancel{\left(\frac{1}{2}\right)^{n}}\cdot n} = \lim_{n\to\infty} \frac{\cancel{n}\left(1+\overset{\to 0}{\frac{1}{n}}\right)\cdot\frac{1}{2}}{\cancel{n}} = \frac{1}{2}$$

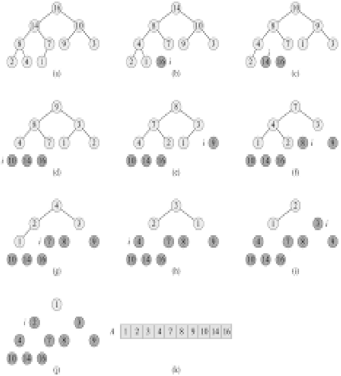


Una max-heap può essere adesso usata efficientemente per progettare un algoritmo di ordinamento. Consideriamo una max-heap, e ricordiamo che una delle proprietá è che il massimo elemento di H si trova in $H[1]$. Se consideriamo $H[1]$ come *giá ordinato* (basta metterlo nella giusta posizione: l'ultima), e sostituiamo il contenuto di $H[1]$, succede che $H[2]$ e $H[3]$ sono ancora max-heap. Quindi chiamando *MaxHeapify* rispettiamo le ipotesi e possiamo ripetere il processo. Il codice di *HeapSort* si basa precisamente su questa osservazione.

Ordinamento con *HeapSort*

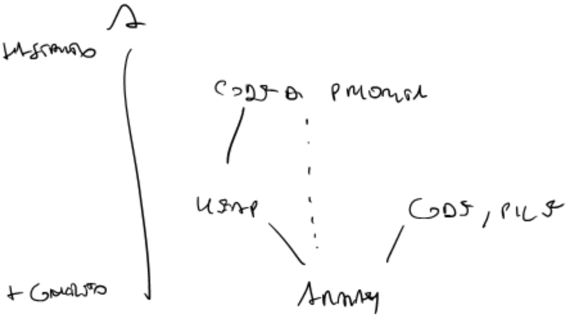
proc ***HeapSort*** (H)

$$\begin{cases} \mathit{BuildMaxHeap}(H) \\ \textbf{for } (i = H.\mathit{length} \textbf{ downto } 2) \begin{cases} \mathit{SwapValue}(H, i, 1) \\ H.\mathit{heapsize} = H.\mathit{heapsize} - 1 \\ \mathit{MaxHeapify}(H, 1) \end{cases} \end{cases}$$



Correttezza e complessità di *HeapSort*

Nel caso di *HeapSort* la **correttezza** è immediata, perchè dipende direttamente dalla correttezza delle procedure su cui è basato. Anche la **terminazione** è ovvia. La **complessità** di *HeapSort*, nel caso peggiore, si calcola come segue. La chiamata a *BuildMaxHeap* costa $\Theta(n)$; per ogni i (cioè n volte) si effettua uno scambio ($O(1)$) ed una chiamata a *MaxHeapify* ($\Theta(log(n))$). Il totale è $\Theta(n \cdot log(n))$. La complessità è la stessa nel caso migliore e quindi nel caso medio: dopo aver effettuato *BuildMaxHeap*, per definizione ogni chiamata successiva a *MaxHeapify* (dopo lo scambio) deve arrivare alle foglie. Possiamo anche osservare che la nostra implementazione di *HeapSort* non è stabile: si può dimostrare osservando il suo comportamento sull'array $H = [1, 1]$. D'altra parte è certamente in place, a meno delle chiamate ricorsive, che, come abbiamo osservato, sono unicamente tail-ricorsive.


+ ESPECÍFICO
COLA DE PRIORIDAD
HEAP
COLA, PILA
+ GENÉRICO
ARRAY

Dinamica

Una **coda di prioritá** è una struttura dati astratta basata sull'ordinamento, e necessariamente compatta. Possiamo costruire una coda di prioritá basandoci su una min-heap. A differenza di una coda classica, che implementa una politica FIFO, una coda di priorità associa ad ogni elemento una chiave, la **priorità**, e serve (cioè estrae) l'elemento a priorità più bassa. Questa estrazione è associata all'operazione che **aggiusta** la struttura dati, ed anche alla possibilità di **inserire** nuovi elementi, o **cambiare la priorità** di un elemento inserito. Sia quindi Q una min-heap senza campi aggiuntivi.

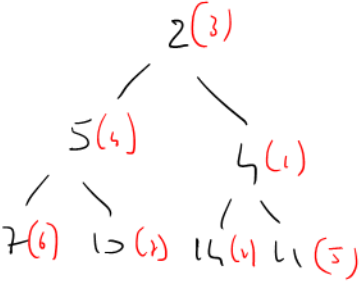
proc ***Enqueue*** $(Q, priority)$
- **if** $(Q.heapsize = Q.length)$
  - **then return** *"overflow"*
- $Q.heapsize = Q.hepsize + 1$
- $Q[heapsize] = \infty$
- $DecreaseKey(Q, Q.heapsize, priority)$

??

proc ***DecreaseKey*** $(Q, i, priority)$
- **if** $(priority > Q[i])$
  - **then return** *"error"*
- $Q[i] = priority$
- **while** $((i > 1)$ **and** $(Q[Parent(i)] > Q[i]))$
  - $SwapValue(Q, i, Parent(i))$
  - $i = Parent(i)$

proc ***ExtractMin*** (Q)
- **if** $(Q.heapsize < 1)$
  - **then return** *"underflow"*
- $min = Q[1]$
- $Q[1] = Q[Q.heapsize]$
- $Q.heapsize = Q.heapsize - 1$
- $MinHeapify(Q, 1)$
- **return** *min*

PROBLEMA HEAP-POSIZIONI



H:

	3	4	1	6	7	2	5
H:	2	5	4	7	10	14	11
	1	2	3	4	5	6	7

P:

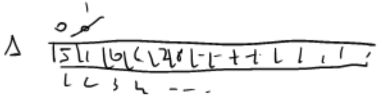
P:	3	6	1	2	7	4	5
	1	2	3	4	5	6	7

La **correttezza** di queste operazioni è immediata da dimostrare usando gli stessi ragionamenti visti prima, cosí come la **terminazione**; nello stesso modo è immediato costruire una versione simmetrica sia dell'estrazione (quindi, del massimo), sia del decremento (quindi, incremento) di una chiave. La **complessità** di tutte queste operazioni è $\Theta(log(n))$ nel caso pessimo.

Code di prioritá su array

Le code di prioritá come struttura dati astratta possono essere implementate anche su array direttamente, senza passare dalla struttura intermedia delle heap. Questa soluzione ha il vantaggio della semplicitá di implementazione. Le complessitá delle operazioni, invece, non sono comparabili. Questo significa che questa soluzione è migliore in qualche caso e peggiore in qualche altro. Nelle code di prioritá che abbiamo visto, ci siamo concentrati solo sulle prioritá, senza porre l'accento sulle chiavi associate. In questa soluzione è conveniente esplicitarle per evitare confusioni.

Code di prioritá su array



Cosi come abbiamo fatto con altre strutture astratte, quindi, diciamo che Q è un array (che immaginiamo sempre con un campo $Q.length$), e diciamo che ogni posizione i ha tre valori: i stesso (la chiave), $Q[i]$ (la sua priorità), e $Q[i].empty$ (che ci indica se l'elemento a chiave i è stato, o no, virtualmente cancellato), e immaginiamo di aver inizializzato tutti i valori $Q[i].empty$ a falso (partiamo da una situazione in cui ci sono tutti). In questo modo, abbiamo fatto l'equivalente della costruzione della coda: ogni elemento (chiave) ha la sua prioritá e si trova dentro la coda. Il **costo** di questa inizializzazione, il cui codice non vediamo, è ovviamente di $\Theta(n)$.

proc ***Enqueue*** $(Q, i, priority)$
- **if** $(i > Q.length)$
  - **then return** *"overflow"*
- $Q[i] = priority$

proc ***DecreaseKey*** $(Q, i, priority)$
- **if** $((Q[i] < priority)$ **or** $(Q[i].empty = 1))$
  - **then return** *"error"*
- $Q[i] = priority$

proc ***ExtractMin*** (Q)
- $MinIndex = 0$
- $MinPriority = \infty$
- **for** $(i = 1$ **to** $Q.length)$
  - **if** $((Q[i] < MinPriority)$ **and** $(Q[i].empty = 0))$
    - **then**
      - $MinPriority = Q[i]$
      - $MinIndex = i$
- **if** $(MinIndex = 0)$
  - **then return** *"underflow"*
- $Q[MinIndex].empty = 1$
- **return** *MinIndex*

Code di prioritá su array

In questa soluzione, la cui **correttezza** e **terminazione** sono immediate, dunque, l'operazione di estrazione del minimo costa $O(n)$, perchè nel caso peggiore il ciclo **for** deve scorrere su tutti gli elementi. Invece, grazie all'ipotesi sul valore degli elementi (che sono naturali), il decremento costa $\Theta(1)$, cosí come l'operazione di inserimento di un nuovo elemento.

Code di prioritá su array vs su heaps: confronto

	array (c.peggiore e medio)	heaps (c. peggiore e medio)
inizial.	$\Theta(n)$	$\Theta(n)$
inserimento	$\Theta(1)$	$\Theta(log(n))$
decremento	$\Theta(1)$	$\Theta(log(n))$
estr. minimo	$\Theta(n)$	$\Theta(log(n))$

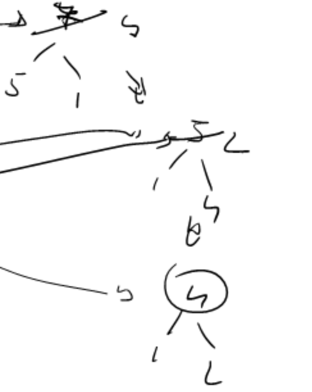
Conclusione

Le heaps sono strutture dati semplici con una grande quantitá di usi, oltre a quelli che abbiamo visto. In molti esercizi, anche tra quelli proposti, la soluzione potrebbe passare attraverso l'uso di heaps anche quando non esplicitamente detto nell'esercizio stesso.

ESERCIZIO $\boxed{68}$

PRENDO I PRIMI k
NE FACCIO UNA MAX-HEAP

A: | 5 | 1 | 7 | 4 | 10 | 2 | 9 |



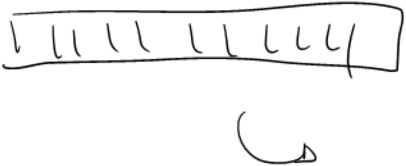
A: | 1 | 2 | 4 | 5 | 7 | 9 | 10 |

$k=3$

$$\Theta(k) + (n-k)(\lg(n)) = \Theta\left(k+(n-k)\lg(n)\right)$$

Inserción BS

$\boxed{k}$



$\Theta(k) + k \cdot \lg(n)$

$\Rightarrow O(k + k\lg(n))$