Algoritmi e Strutture Dati

Heap e Code con Priorità

Alberto Montresor and Davide Rossi

Università di Bologna

9 dicembre 2024

Sommario

Heap

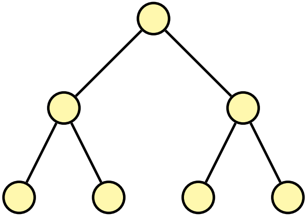
Storia

- Struttura dati inventata da J. Williams nel 1964
- Utilizzata per implementare un nuovo algoritmo di ordinamento: HeapSort
- Williams intuì subito che poteva essere usata per altri scopi
- Seguiamo l'approccio storico nel presentare gli heap
    - Prima HeapSort
    - Poi Code con priorità

Alberi binari

Albero binario perfetto

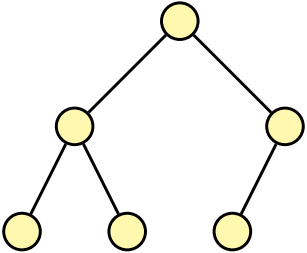
- Tutte le foglie hanno la stessa profondità h
- Nodi interni hanno tutti grado 2
- Dato il numero di nodi n, ha altezza $h = \lfloor \log n \rfloor$
- Dato l'altezza h, ha numeri di nodi $n = 2^{h+1} - 1$



Alberi binari

Albero binario completo

- Tutte le foglie hanno profondità h o $h-1$
- Tutti i nodi a livello h sono "accatastati" a sinistra
- Tutti i nodi interni hanno grado 2, eccetto al più uno
- Dato il numero di nodi n, ha altezza $h = \lfloor \log n \rfloor$



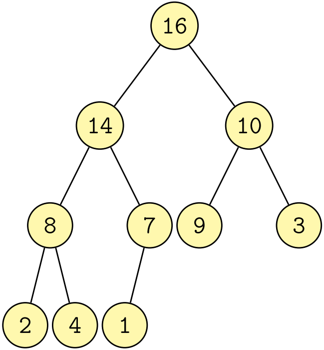
Alberi binari heap

Proprietà heap

Un albero max-heap (min-heap) è un albero binario completo tale che il valore memorizzato in ogni nodo è maggiore (minore) dei valori memorizzati nei suoi figli.

Note

Le definizioni e gli algoritmi per alberi max-heap sono simmetrici rispetto agli algoritmi per alberi min-heap



Alberi binari heap

- Un albero heap non impone una relazione di ordinamento totale fra i figli di un nodo
- Un albero heap è un ordinamento parziale
  - Riflessivo: Ogni nodo è $\geq$ di se stesso
  - Antisimmetrico: se $n \geq m$ e $m \geq n$, allora $m = n$
  - Transitivo: se $n \geq m$ e $m \geq r$, allora $n \geq r$
- Ordinamenti parziali
  - Nozione più debole di un ordinamento totale...
  - ... ma più semplice da costruire

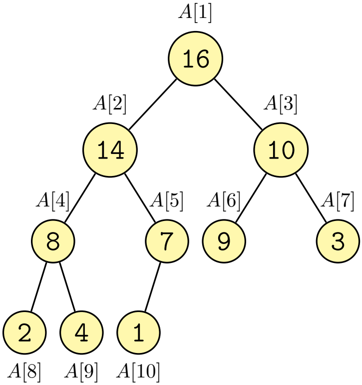
Alberi binari heap

Vettore heap

Un albero heap può essere rappresentato tramite un vettore heap

Memorizzazione ($A[1 \ldots n]$)

Radice	$root() = 1$
Padre nodo i	$p(i) = \lfloor i/2 \rfloor$
Figlio sx nodo i	$l(i) = 2i$
Figlio dx nodo i	$r(i) = 2i + 1$



$A[1]$	$A[2]$	$A[3]$	$A[4]$	$A[5]$	$A[6]$	$A[7]$	$A[8]$	$A[9]$	$A[10]$
16	14	10	8	7	9	3	2	4	1

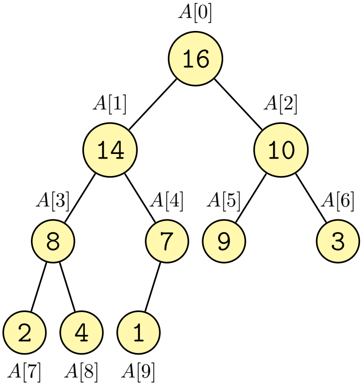
Alberi binari heap

Vettore heap

Un albero heap può essere rappresentato tramite un vettore heap

Memorizzazione ($A[0 \ldots n-1]$)

Radice	$root() = 0$
Padre nodo i	$p(i) = \lfloor (i-1)/2 \rfloor$
Figlio sx nodo i	$l(i) = 2i + 1$
Figlio dx nodo i	$r(i) = 2i + 2$



$A[0]$	$A[1]$	$A[2]$	$A[3]$	$A[4]$	$A[5]$	$A[6]$	$A[7]$	$A[8]$	$A[9]$
16	14	10	8	7	9	3	2	4	1

Alberi binari heap

Proprietà max-heap su vettore

$$A[i] \geq A[l(i)], A[i] \geq A[r(i)]$$

Proprietà min-heap su vettore

$$A[i] \leq A[l(i)], A[i] \leq A[r(i)]$$

HeapSort

Organizzazione heapsort()

Ordina un max-heap "in-place", prima costruendo un max-heap nel vettore e poi spostando l'elemento max in ultima posizione, ripristinando la proprietà max-heap

- heapBuild()
  Costruisce un max-heap a partire da un vettore non ordinato
- maxHeapRestore()
  Ripristina la proprietà max-heap

maxHeapRestore()

Input

Un vettore A e un indice i, tale per cui gli alberi binari con radici $l(i)$ e $r(i)$ sono max-heap

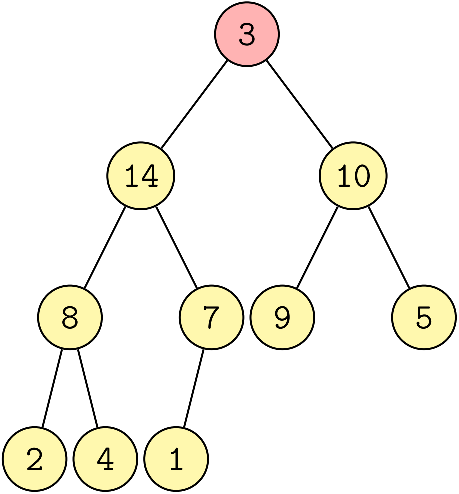
Osservazione

- È possibile che $A[i]$ sia minore di $A[l(i)]$ o $A[r(i)]$
- In altre parole, non è detto che il sottoalbero con radice i sia un max-heap

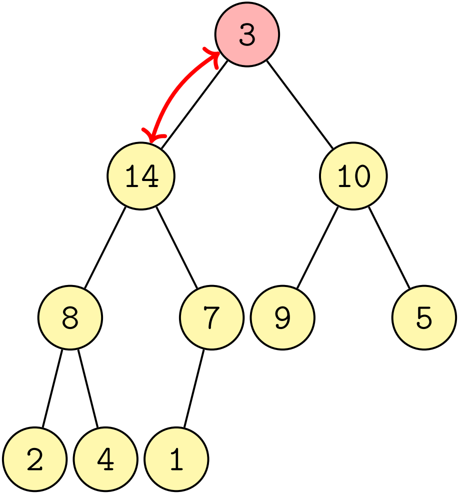
Goal

Modificare in-place il vettore A in modo tale che l'albero binario con radice i sia un max-heap

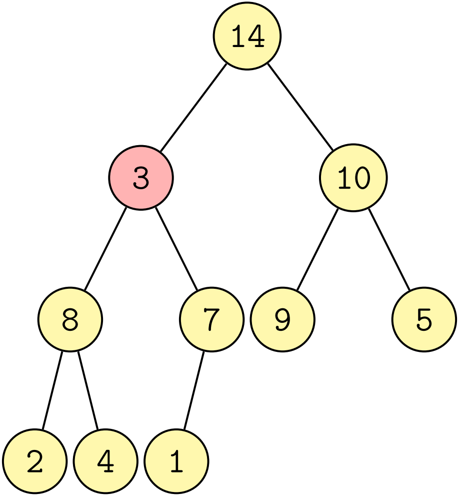
Esempio



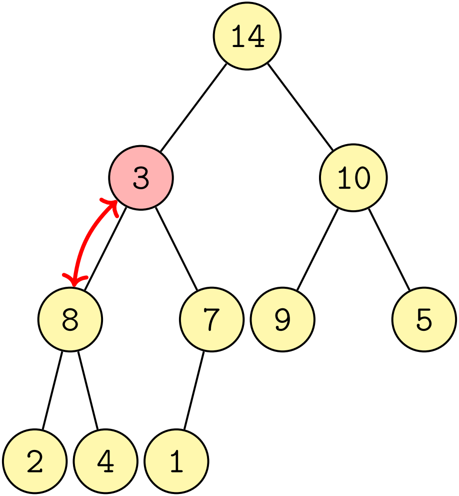
Esempio



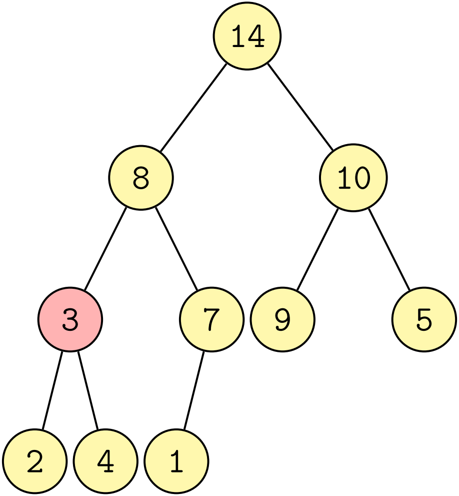
Esempio



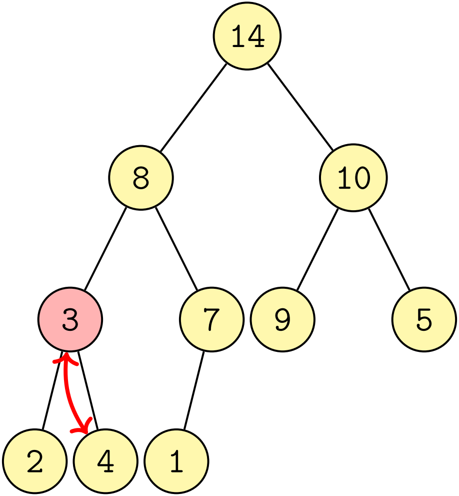
Esempio



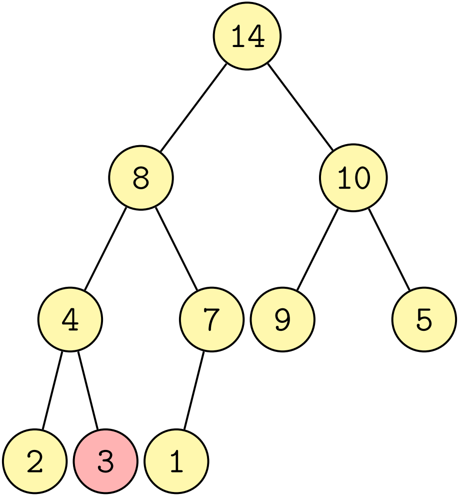
Esempio



Esempio



Esempio



Ripristinare la proprietà max-heap

maxHeapRestore(ITEM[] A, **int** i, **int** dim)

int $max = i$
if $l(i) \leq dim$ **and** $A[l(i)] > A[max]$ **then**
  $max = l(i)$
if $r(i) \leq dim$ **and** $A[r(i)] > A[max]$ **then**
  $max = r(i)$
if $i \neq max$ **then**
  swap(A, i, max)
  maxHeapRestore(A, max, dim)

Qual è la complessità computazionale di maxHeapRestore()?

maxHeapRestore() – Complessità computazionale

Qual è la complessità computazionale di maxHeapRestore()?

- Ad ogni chiamata, vengono eseguiti $O(1)$ confronti
- Se il nodo i non è massimo, si richiama ricorsivamente maxHeapRestore() su uno dei figli
- L'esecuzione termina quando si raggiunge una foglia
- L'altezza dell'albero è pari a $\lfloor \log n \rfloor$

Complessità

$$T(n) = O(\log n)$$

Dimostrazione correttezza (per induzione sull'altezza)

Teorema

Al termine dell'esecuzione, l'albero radicato in $A[i]$ rispetta la proprietà max-heap

Dimostrazione correttezza (per induzione sull'altezza)

Teorema

Al termine dell'esecuzione, l'albero radicato in $A[i]$ rispetta la proprietà max-heap

Caso base: altezza $h = 0$

Se $h = 0$, l'albero è dato da un solo nodo che rispetta la proprietà heap

Ipotesi induttiva

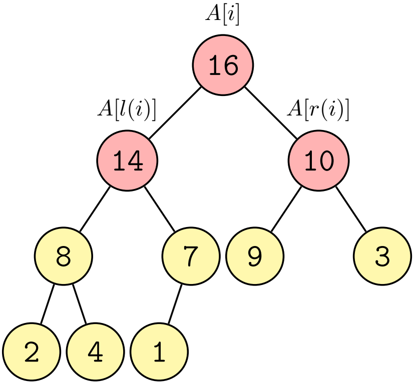
L'algoritmo funziona correttamente su tutti gli alberi di altezza minore di h

Dimostrazione correttezza (per induzione sull'altezza)

Induzione - Altezza h - Caso 1

$A[i] \geq A[l(i)], A[i] \geq A[r(i)]$:

- L'albero radicato in $A[i]$ rispetta la proprietà max-heap
- L'algoritmo termina (CVD)

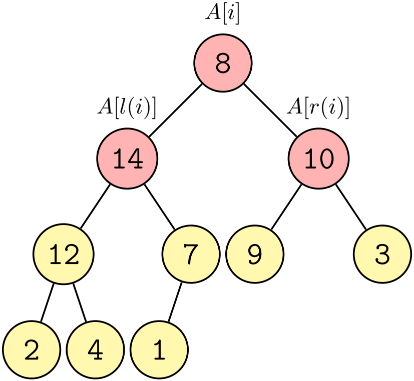


Dimostrazione correttezza (per induzione sull'altezza)

Induzione - Altezza h - Caso 2

$A[l(i)] > A[i], A[l(i)] > A[r(i)]$:

- Viene fatto uno scambio $A[i] \leftrightarrow A[l(i)]$
- Dopo lo scambio, $A[i] \geq A[l(i)], A[i] \geq A[r(i)]$
- Il sottoalbero $A[r(i)]$ è inalterato e rispetta la proprietà heap
- Il sottoalbero $A[l(i)]$ può aver perso la proprietà heap
- Si applica maxHeapRestore() ricorsivamente su di $A[l(i)]$, che ha altezza minore di h



Passo induttivo - Caso 3

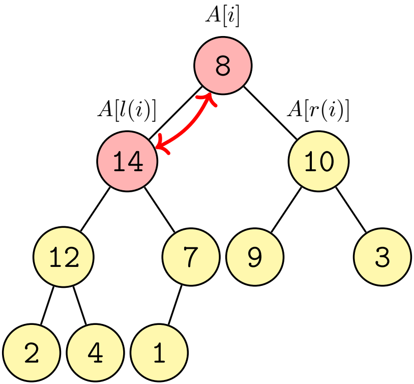
Simmetrico rispetto al Caso 2

Dimostrazione correttezza (per induzione sull'altezza)

Induzione - Altezza h - Caso 2

$A[l(i)] > A[i], A[l(i)] > A[r(i)]$:

- Viene fatto uno scambio $A[i] \leftrightarrow A[l(i)]$
- Dopo lo scambio, $A[i] \geq A[l(i)], A[i] \geq A[r(i)]$
- Il sottoalbero $A[r(i)]$ è inalterato e rispetta la proprietà heap
- Il sottoalbero $A[l(i)]$ può aver perso la proprietà heap
- Si applica maxHeapRestore() ricorsivamente su di $A[l(i)]$, che ha altezza minore di h



Passo induttivo - Caso 3

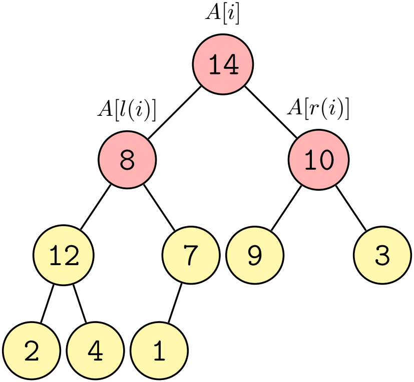
Simmetrico rispetto al Caso 2

Dimostrazione correttezza (per induzione sull'altezza)

Induzione - Altezza h - Caso 2

$A[l(i)] > A[i], A[l(i)] > A[r(i)]$:

- Viene fatto uno scambio $A[i] \leftrightarrow A[l(i)]$
- Dopo lo scambio, $A[i] \geq A[l(i)], A[i] \geq A[r(i)]$
- Il sottoalbero $A[r(i)]$ è inalterato e rispetta la proprietà heap
- Il sottoalbero $A[l(i)]$ può aver perso la proprietà heap
- Si applica maxHeapRestore() ricorsivamente su di $A[l(i)]$, che ha altezza minore di h



Passo induttivo - Caso 3

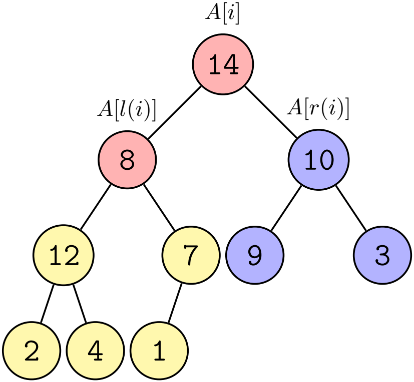
Simmetrico rispetto al Caso 2

Dimostrazione correttezza (per induzione sull'altezza)

Induzione - Altezza h - Caso 2

$A[l(i)] > A[i], A[l(i)] > A[r(i)]$:

- Viene fatto uno scambio $A[i] \leftrightarrow A[l(i)]$
- Dopo lo scambio, $A[i] \geq A[l(i)], A[i] \geq A[r(i)]$
- Il sottoalbero $A[r(i)]$ è inalterato e rispetta la proprietà heap
- Il sottoalbero $A[l(i)]$ può aver perso la proprietà heap
- Si applica maxHeapRestore() ricorsivamente su di $A[l(i)]$, che ha altezza minore di h



Passo induttivo - Caso 3

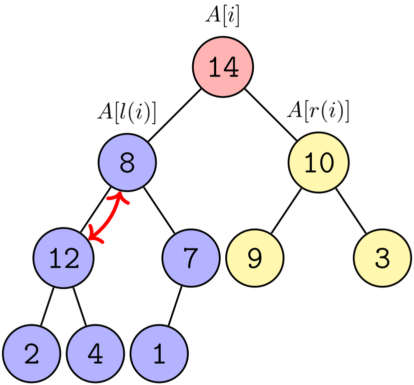
Simmetrico rispetto al Caso 2

Dimostrazione correttezza (per induzione sull'altezza)

Induzione - Altezza h - Caso 2

$A[l(i)] > A[i], A[l(i)] > A[r(i)]$:

- Viene fatto uno scambio $A[i] \leftrightarrow A[l(i)]$
- Dopo lo scambio, $A[i] \geq A[l(i)], A[i] \geq A[r(i)]$
- Il sottoalbero $A[r(i)]$ è inalterato e rispetta la proprietà heap
- Il sottoalbero $A[l(i)]$ può aver perso la proprietà heap
- Si applica maxHeapRestore() ricorsivamente su di $A[l(i)]$, che ha altezza minore di h



Passo induttivo - Caso 3

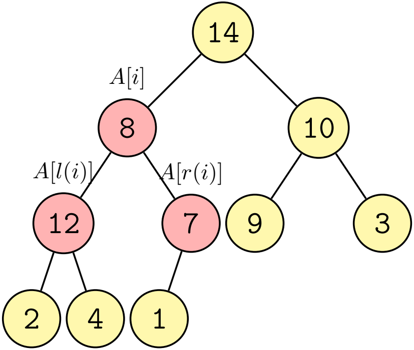
Simmetrico rispetto al Caso 2

Dimostrazione correttezza (per induzione sull'altezza)

Induzione - Altezza h - Caso 2

$A[l(i)] > A[i], A[l(i)] > A[r(i)]$:

- Viene fatto uno scambio $A[i] \leftrightarrow A[l(i)]$
- Dopo lo scambio, $A[i] \geq A[l(i)], A[i] \geq A[r(i)]$
- Il sottoalbero $A[r(i)]$ è inalterato e rispetta la proprietà heap
- Il sottoalbero $A[l(i)]$ può aver perso la proprietà heap
- Si applica maxHeapRestore() ricorsivamente su di $A[l(i)]$, che ha altezza minore di h



Passo induttivo - Caso 3

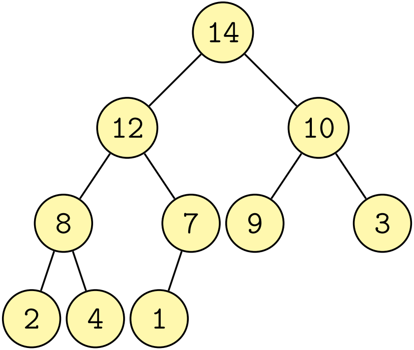
Simmetrico rispetto al Caso 2

Dimostrazione correttezza (per induzione sull'altezza)

Induzione - Altezza h - Caso 2

$A[l(i)] > A[i], A[l(i)] > A[r(i)]$:

- Viene fatto uno scambio $A[i] \leftrightarrow A[l(i)]$
- Dopo lo scambio, $A[i] \geq A[l(i)], A[i] \geq A[r(i)]$
- Il sottoalbero $A[r(i)]$ è inalterato e rispetta la proprietà heap
- Il sottoalbero $A[l(i)]$ può aver perso la proprietà heap
- Si applica maxHeapRestore() ricorsivamente su di $A[l(i)]$, che ha altezza minore di h



Passo induttivo - Caso 3

Simmetrico rispetto al Caso 2

heapBuild()

Principio di funzionamento

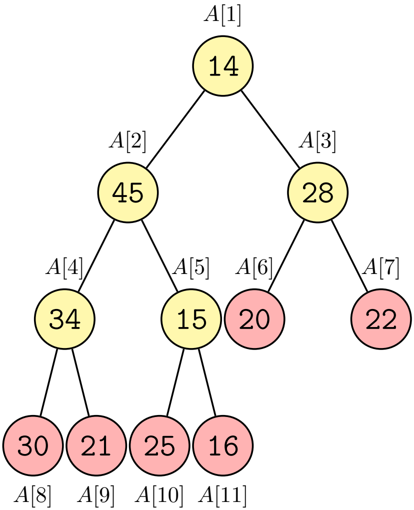
- Sia $A[1 \dots n]$ un vettore da ordinare
- Tutti i nodi $A[\lfloor n/2 \rfloor + 1 \dots n]$ sono foglie dell'albero e quindi heap contenenti 1 elemento
- La procedura heapBuild()
  - attraversa i restanti nodi dell'albero, a partire da $\lfloor n/2 \rfloor$ fino ad 1
  - esegue maxHeapRestore() su ognuno di essi

heapBuild(Item[] A, **int** n)

for $i = \lfloor n/2 \rfloor$ **downto** 1 **do**
    maxHeapRestore(A, i, n)

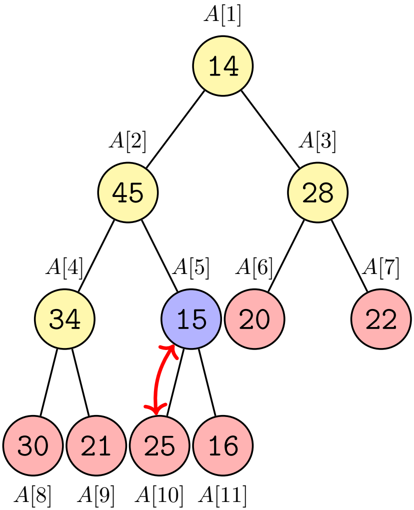
Esempio

- I nodi $A[\lfloor n/2 \rfloor + 1 \ldots n]$ sono foglie dell'albero e quindi heap di un elemento
- Per ogni posizione da $\lfloor n/2 \rfloor$ fino ad 1, si esegue maxHeapRestore()



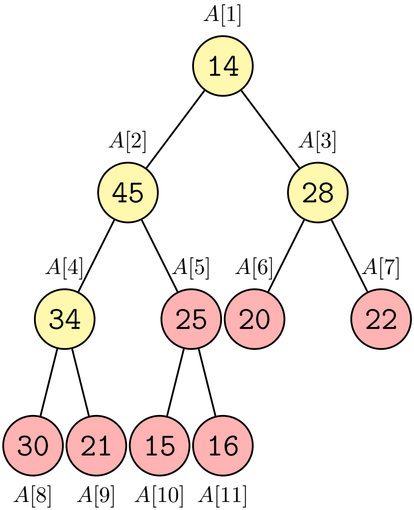
Esempio

- I nodi $A[\lfloor n/2 \rfloor + 1 \ldots n]$ sono foglie dell'albero e quindi heap di un elemento
- Per ogni posizione da $\lfloor n/2 \rfloor$ fino ad 1, si esegue maxHeapRestore()



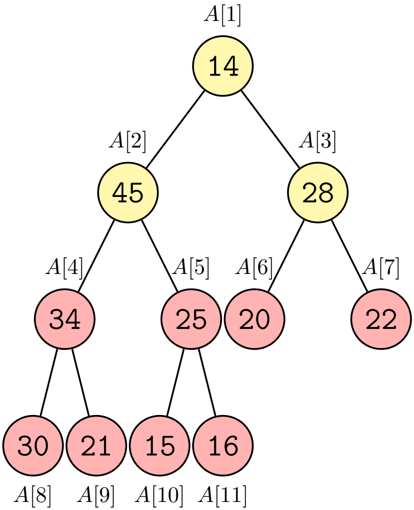
Esempio

- I nodi $A[\lfloor n/2 \rfloor + 1 \ldots n]$ sono foglie dell'albero e quindi heap di un elemento
- Per ogni posizione da $\lfloor n/2 \rfloor$ fino ad 1, si esegue maxHeapRestore()



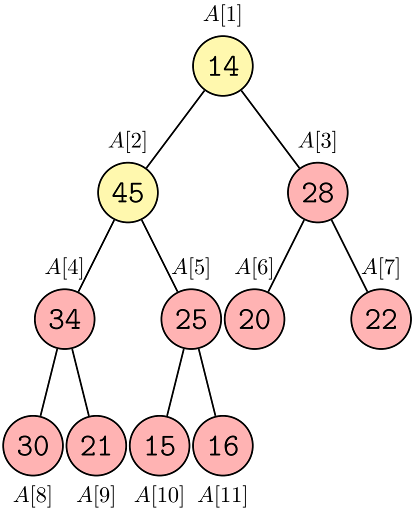
Esempio

- I nodi $A[\lfloor n/2 \rfloor + 1 \ldots n]$ sono foglie dell'albero e quindi heap di un elemento
- Per ogni posizione da $\lfloor n/2 \rfloor$ fino ad 1, si esegue maxHeapRestore()



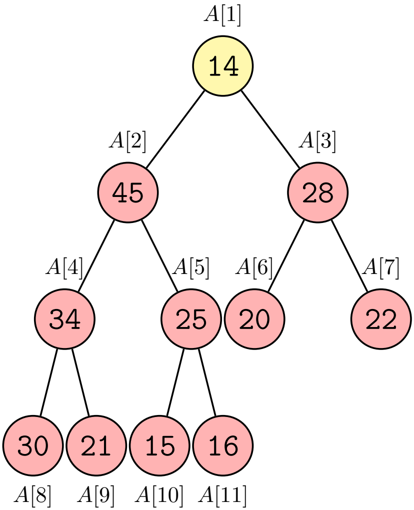
Esempio

- I nodi $A[\lfloor n/2 \rfloor + 1 \ldots n]$ sono foglie dell'albero e quindi heap di un elemento
- Per ogni posizione da $\lfloor n/2 \rfloor$ fino ad 1, si esegue maxHeapRestore()



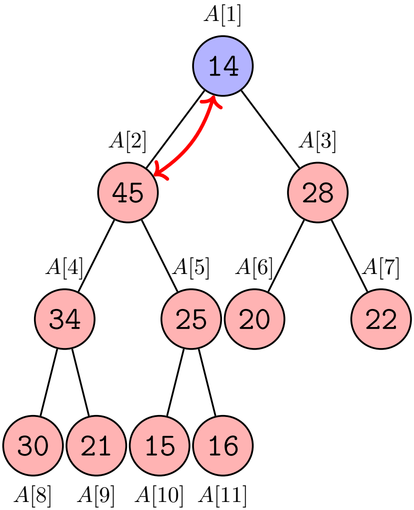
Esempio

- I nodi $A[\lfloor n/2 \rfloor + 1 \ldots n]$ sono foglie dell'albero e quindi heap di un elemento
- Per ogni posizione da $\lfloor n/2 \rfloor$ fino ad 1, si esegue maxHeapRestore()



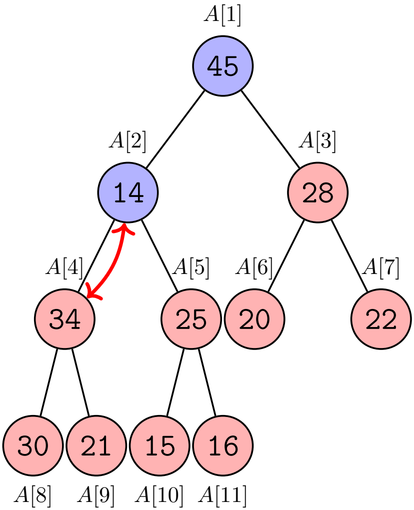
Esempio

- I nodi $A[\lfloor n/2 \rfloor + 1 \ldots n]$ sono foglie dell'albero e quindi heap di un elemento
- Per ogni posizione da $\lfloor n/2 \rfloor$ fino ad 1, si esegue maxHeapRestore()



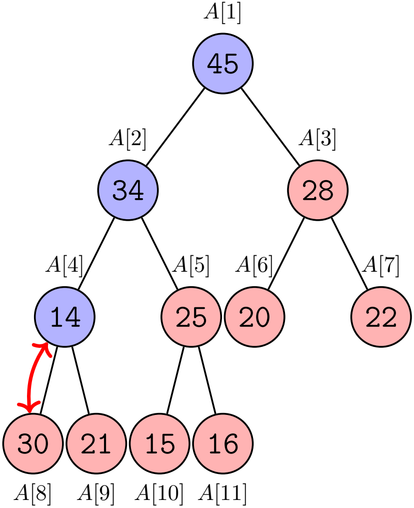
Esempio

- I nodi $A[\lfloor n/2 \rfloor + 1 \ldots n]$ sono foglie dell'albero e quindi heap di un elemento
- Per ogni posizione da $\lfloor n/2 \rfloor$ fino ad 1, si esegue maxHeapRestore()



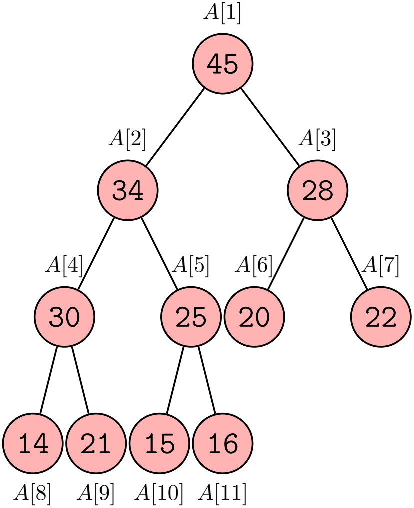
Esempio

- I nodi $A[\lfloor n/2 \rfloor + 1 \ldots n]$ sono foglie dell'albero e quindi heap di un elemento
- Per ogni posizione da $\lfloor n/2 \rfloor$ fino ad 1, si esegue maxHeapRestore()



Esempio

- I nodi $A[\lfloor n/2 \rfloor + 1 \ldots n]$ sono foglie dell'albero e quindi heap di un elemento
- Per ogni posizione da $\lfloor n/2 \rfloor$ fino ad 1, si esegue maxHeapRestore()



Correttezza

Invariante di ciclo

All'inizio di ogni iterazione del ciclo **for**, i nodi $[i+1,\ldots,n]$ sono radice di uno heap.

Dimostrazione – Inizializzazione

- All'inizio, $i = \lfloor n/2 \rfloor$.
- Supponiamo che $\lfloor n/2 \rfloor + 1$ non sia una foglia
- Quindi ha almeno il figlio sinistro: $2\lfloor n/2 \rfloor + 2$
- Questo ha indice $n+1$ oppure $n+2$, assurdo perché n è la dimensione massima
- La dimostrazione vale per tutti gli indici successivi

Correttezza

Invariante di ciclo

All'inizio di ogni iterazione del ciclo **for**, i nodi $[i + 1, \ldots, n]$ sono radice di uno heap.

Dimostrazione – Conservazione

- È possibile applicare maxHeapRestore al nodo i, perché $2i < 2i + 1 \leq n$ sono entrambi radici di heap
- Al termine dell'iterazione, tutti i nodi $[i \ldots n]$ sono radici di heap

Dimostrazione – Conclusione

- Al termine, $i = 0$. Quindi il nodo 1 è radice di uno heap.

Complessità

heapBuild(ITEM[] A, **int** n)

for $i = \lfloor n/2 \rfloor$ **downto** 1 **do**
  maxHeapRestore(A, i, n)

Qual è la complessità di HEAPBUILD()?

Complessità

$\mathsf{heapBuild}(\textsc{Item}[\,]\ A,\ \textbf{int}\ n)$

---

for $i = \lfloor n/2 \rfloor$ **downto** 1 **do**
  $\mathsf{maxHeapRestore}(A, i, n)$

Qual è la complessità di HEAPBUILD()?

- Limite superiore:

Complessità

heapBuild(ITEM[] A, **int** n)

for $i = \lfloor n/2 \rfloor$ **downto** 1 **do**
  maxHeapRestore(A, i, n)

Qual è la complessità di HEAPBUILD()?

- Limite superiore: $T(n) = O(n \log n)$
- Limite inferiore:

Complessità

---

heapBuild(ITEM[] A, **int** n)

---

for $i = \lfloor n/2 \rfloor$ **downto** 1 **do**
  maxHeapRestore(A, i, n)

Qual è la complessità di HEAPBUILD()?

- Limite superiore: $T(n) = O(n \log n)$
- Limite inferiore: $T(n) = \Omega(n \log n)$?

Complessità

Le operazioni maxHeapRestore() vengono eseguite un numero decrescente di volte su heap di altezza crescente

Altezza	**# Volte**
0	$\lfloor n/2 \rfloor$
1	$\lfloor n/4 \rfloor$
2	$\lfloor n/8 \rfloor$
$\cdots$	$\cdots$
h	$\lfloor n/2^{h+1} \rfloor$

$$
\begin{aligned}
T(n) &\leq \sum_{h=1}^{\lfloor \log n \rfloor} \frac{n}{2^{h+1}} h \\
&= n \sum_{h=1}^{\lfloor \log n \rfloor} \left(\frac{1}{2}\right)^{h+1} h \\
&= n/2 \sum_{h=1}^{\lfloor \log n \rfloor} \left(\frac{1}{2}\right)^{h} h \\
&\leq n/2 \sum_{h=1}^{+\infty} \left(\frac{1}{2}\right)^{h} h \qquad = n = O(n)
\end{aligned}
$$

Formula: $\displaystyle\sum_{h=1}^{+\infty} hx^h = \frac{x}{(1-x)^2}$, per $x < 1$

heapSort()

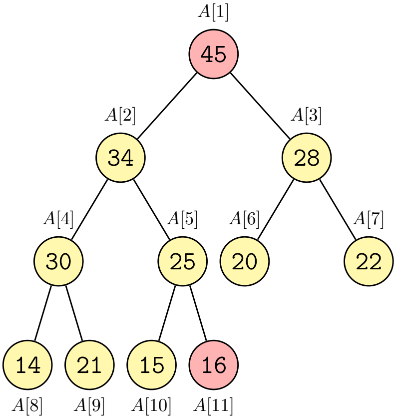
Principio di funzionamento

- L'elemento in prima posizione contiene il massimo
- Viene collocato in fondo
- L'elemento in fondo viene spostato in testa
- Si chiama maxHeapRestore() per ripristinare la situazione
- La dimensione dello heap viene progressivamente ridotta (indice i)

```
heapSort(Item[] A, int n)

heapBuild(A, n)
for i = n downto 2 do
    swap(A, 1, i)
    maxHeapRestore(A, 1, i − 1)
```

Esempio



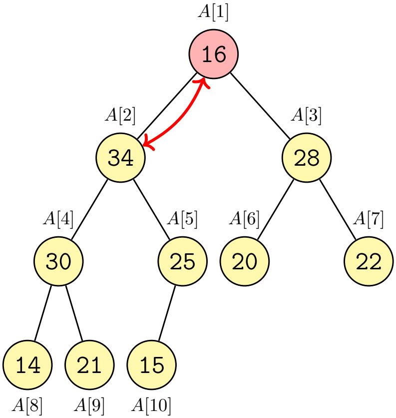
A[1]	A[2]	A[3]	A[4]	A[5]	A[6]	A[7]	A[8]	A[9]	A[10]	A[11]
45	34	28	30	25	20	22	14	21	15	16

HEAPSORT(ITEM[] A, **int** n)

---

heapBuild(A, n)
for $i = n$ **downto** 2 **do**
  swap$(A, 1, i)$
  maxHeapRestore$(A, 1, i-1)$

Esempio

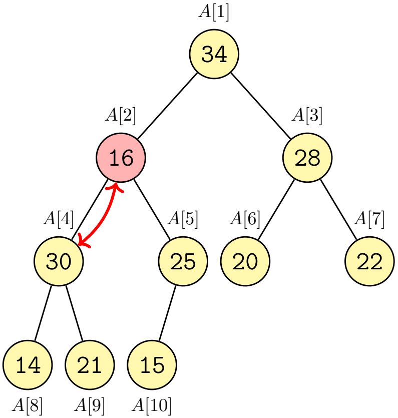


A[1]	A[2]	A[3]	A[4]	A[5]	A[6]	A[7]	A[8]	A[9]	A[10]	A[11]
16	34	28	30	25	20	22	14	21	15	45

heapSort(Item[] A, **int** n)

heapBuild(A, n)
for $i = n$ **downto** 2 **do**
  swap$(A, 1, i)$
  maxHeapRestore$(A, 1, i-1)$

Esempio

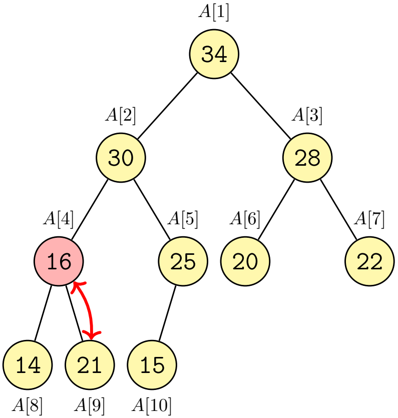


A[1]	A[2]	A[3]	A[4]	A[5]	A[6]	A[7]	A[8]	A[9]	A[10]	A[11]
34	16	28	30	25	20	22	14	21	15	45

HEAPSORT(ITEM[ ] A, **int** n)

heapBuild(A, n)
for $i = n$ **downto** 2 **do**
    swap$(A, 1, i)$
    maxHeapRestore$(A, 1, i-1)$

Esempio



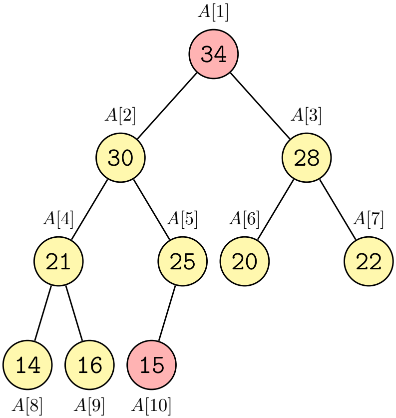
A[1]	A[2]	A[3]	A[4]	A[5]	A[6]	A[7]	A[8]	A[9]	A[10]	A[11]
34	30	28	16	25	20	22	14	21	15	45

HEAPSORT(ITEM[ ] A, **int** n)

---

heapBuild(A, n)
for $i = n$ **downto** 2 **do**
  swap$(A, 1, i)$
  maxHeapRestore$(A, 1, i-1)$

Esempio

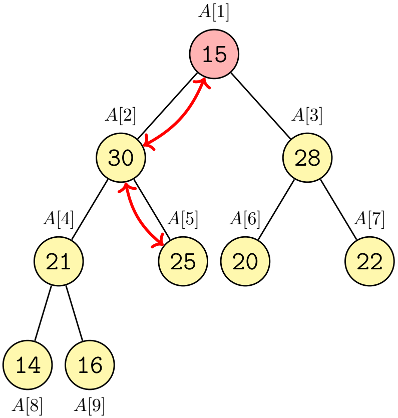


A[1]	A[2]	A[3]	A[4]	A[5]	A[6]	A[7]	A[8]	A[9]	A[10]	A[11]
34	30	28	21	25	20	22	14	16	15	45

$\textsc{heapSort}(\textsc{Item}[\,]\ A, \textbf{int}\ n)$

heapBuild(A, n)
for $i = n$ **downto** 2 **do**
    swap$(A, 1, i)$
    maxHeapRestore$(A, 1, i-1)$

Esempio

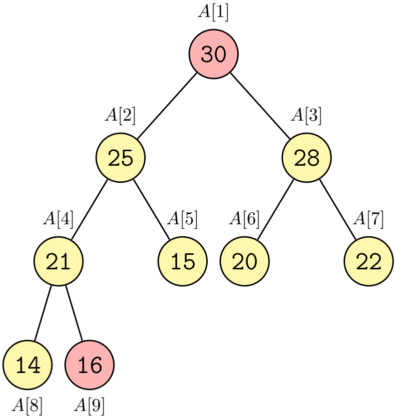


A[1]	A[2]	A[3]	A[4]	A[5]	A[6]	A[7]	A[8]	A[9]	A[10]	A[11]
15	30	28	21	25	20	22	14	16	34	45

HEAPSORT(ITEM[] A, **int** n)

heapBuild(A, n)
for $i = n$ **downto** 2 **do**
  swap($A, 1, i$)
  maxHeapRestore($A, 1, i-1$)

Esempio



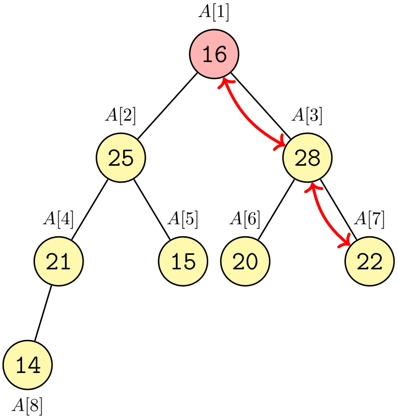
$A[1]$	$A[2]$	$A[3]$	$A[4]$	$A[5]$	$A[6]$	$A[7]$	$A[8]$	$A[9]$	$A[10]$	$A[11]$
30	25	28	21	15	20	22	14	16	34	45

HEAPSORT(ITEM[ ] A, **int** n)

---

heapBuild(A, n)
for $i = n$ **downto** 2 **do**
  swap$(A, 1, i)$
  maxHeapRestore$(A, 1, i - 1)$

Esempio



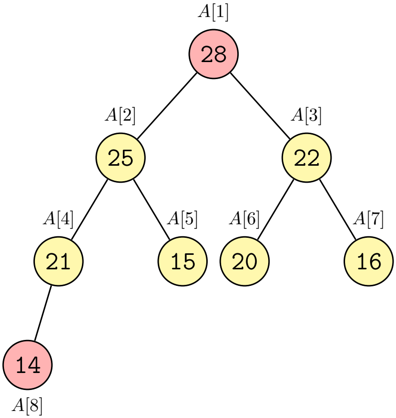
A[1]	A[2]	A[3]	A[4]	A[5]	A[6]	A[7]	A[8]	A[9]	A[10]	A[11]
16	25	28	21	15	20	22	14	30	34	45

HEAPSORT(ITEM[ ] A, **int** n)

---

heapBuild(A, n)
for $i = n$ **downto** 2 **do**
  swap$(A, 1, i)$
  maxHeapRestore$(A, 1, i - 1)$

Esempio

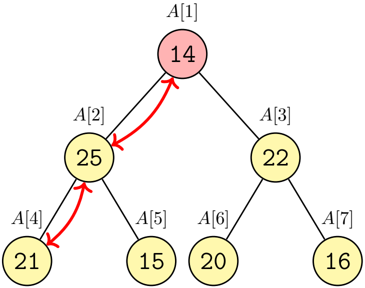


A[1]	A[2]	A[3]	A[4]	A[5]	A[6]	A[7]	A[8]	A[9]	A[10]	A[11]
28	25	22	21	15	20	16	14	30	34	45

HEAPSORT(ITEM[ ] A, **int** n)

heapBuild(A, n)
for $i = n$ **downto** 2 **do**
  swap$(A, 1, i)$
  maxHeapRestore$(A, 1, i - 1)$

Esempio



$\textsc{heapSort}(\textsc{Item}[\,]\ A,\ \textbf{int}\ n)$

---

heapBuild(A, n)

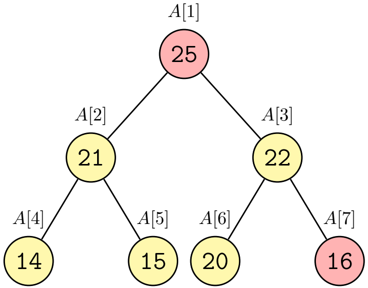
for $i = n$ **downto** 2 **do**

  swap$(A, 1, i)$

  maxHeapRestore$(A, 1, i-1)$

A[1]	A[2]	A[3]	A[4]	A[5]	A[6]	A[7]	A[8]	A[9]	A[10]	A[11]
14	25	22	21	15	20	16	28	30	34	45

Esempio



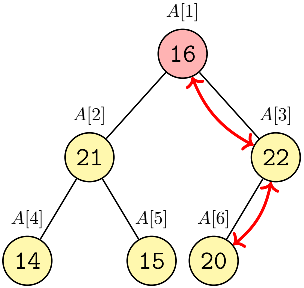
$\textsc{heapSort}(\textsc{Item}[\,]\ A, \textbf{int}\ n)$

---

heapBuild(A, n)
for $i = n$ **downto** 2 **do**
    swap$(A, 1, i)$
    maxHeapRestore$(A, 1, i-1)$

A[1]	A[2]	A[3]	A[4]	A[5]	A[6]	A[7]	A[8]	A[9]	A[10]	A[11]
25	21	22	14	15	20	16	28	30	34	45

Esempio

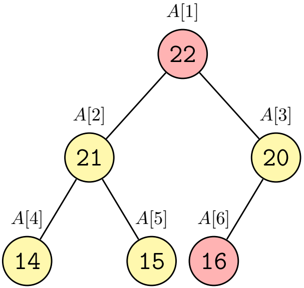


heapSort(Item[] A, **int** n)

heapBuild(A, n)
for $i = n$ **downto** 2 **do**
  swap($A, 1, i$)
  maxHeapRestore($A, 1, i - 1$)

$A[1]$	$A[2]$	$A[3]$	$A[4]$	$A[5]$	$A[6]$	$A[7]$	$A[8]$	$A[9]$	$A[10]$	$A[11]$
16	21	22	14	15	20	25	28	30	34	45

Esempio

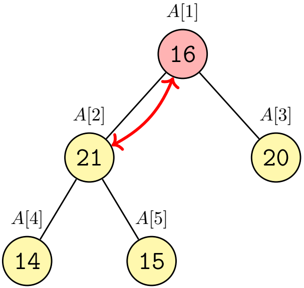


heapSort(Item[] A, **int** n)

heapBuild(A, n)
for $i = n$ **downto** 2 **do**
  swap$(A, 1, i)$
  maxHeapRestore$(A, 1, i-1)$

$A[1]$	$A[2]$	$A[3]$	$A[4]$	$A[5]$	$A[6]$	$A[7]$	$A[8]$	$A[9]$	$A[10]$	$A[11]$
22	21	20	14	15	16	25	28	30	34	45

Esempio



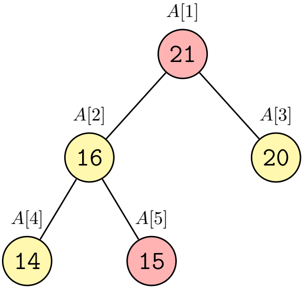
HEAPSORT(ITEM[ ] A, **int** n)

---

heapBuild(A, n)
for $i = n$ **downto** 2 **do**
    swap$(A, 1, i)$
    maxHeapRestore$(A, 1, i - 1)$

A[1]	A[2]	A[3]	A[4]	A[5]	A[6]	A[7]	A[8]	A[9]	A[10]	A[11]
16	21	20	14	15	22	25	28	30	34	45

Esempio

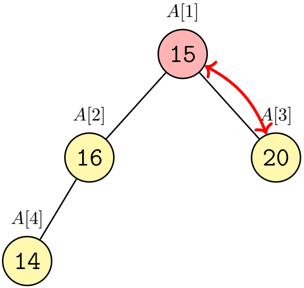


heapSort(Item[] A, **int** n)

heapBuild(A, n)
for $i = n$ **downto** 2 **do**
    swap($A, 1, i$)
    maxHeapRestore($A, 1, i - 1$)

$A[1]$	$A[2]$	$A[3]$	$A[4]$	$A[5]$	$A[6]$	$A[7]$	$A[8]$	$A[9]$	$A[10]$	$A[11]$
21	16	20	14	15	22	25	28	30	34	45

Esempio

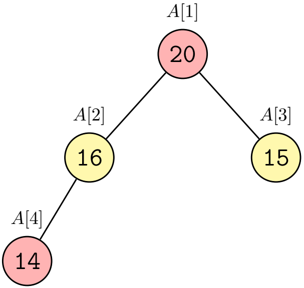


A[1]	A[2]	A[3]	A[4]	A[5]	A[6]	A[7]	A[8]	A[9]	A[10]	A[11]
15	16	20	14	21	22	25	28	30	34	45

heapSort(Item[] A, **int** n)

heapBuild(A, n)
for $i = n$ **downto** 2 **do**
    swap$(A, 1, i)$
    maxHeapRestore$(A, 1, i-1)$

Esempio



$\text{HEAPSORT}(\text{ITEM}[\,]\ A, \textbf{int}\ n)$

---

heapBuild(A, n)

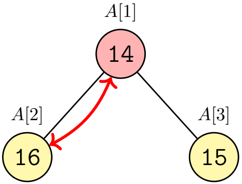
for $i = n$ **downto** 2 **do**

  swap$(A, 1, i)$

  maxHeapRestore$(A, 1, i-1)$

A[1]	A[2]	A[3]	A[4]	A[5]	A[6]	A[7]	A[8]	A[9]	A[10]	A[11]
20	16	15	14	21	22	25	28	30	34	45

Esempio



$\textsc{heapSort}(\textsc{Item}[\,]\ A, \textbf{int}\ n)$

---

heapBuild(A, n)

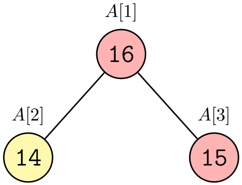
for $i = n$ **downto** 2 **do**

  swap$(A, 1, i)$

  maxHeapRestore$(A, 1, i - 1)$

$A[1]$	$A[2]$	$A[3]$	$A[4]$	$A[5]$	$A[6]$	$A[7]$	$A[8]$	$A[9]$	$A[10]$	$A[11]$
14	16	15	20	21	22	25	28	30	34	45

Esempio



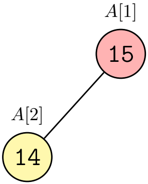
HEAPSORT(ITEM[] A, **int** n)

---

heapBuild(A, n)
for $i = n$ **downto** 2 **do**
    swap$(A, 1, i)$
    maxHeapRestore$(A, 1, i - 1)$

$A[1]$	$A[2]$	$A[3]$	$A[4]$	$A[5]$	$A[6]$	$A[7]$	$A[8]$	$A[9]$	$A[10]$	$A[11]$
16	14	15	20	21	22	25	28	30	34	45

Esempio



heapSort(Item[] A, **int** n)

heapBuild(A, n)
for $i = n$ **downto** 2 **do**
  swap$(A, 1, i)$
  maxHeapRestore$(A, 1, i-1)$

$A[1]$	$A[2]$	$A[3]$	$A[4]$	$A[5]$	$A[6]$	$A[7]$	$A[8]$	$A[9]$	$A[10]$	$A[11]$
15	14	16	20	21	22	25	28	30	34	45

Esempio

$A[1]$

14

heapSort(Item[] A, **int** n)

heapBuild(A, n)
for $i = n$ **downto** 2 **do**
    swap($A, 1, i$)
    maxHeapRestore($A, 1, i - 1$)

$A[1]$	$A[2]$	$A[3]$	$A[4]$	$A[5]$	$A[6]$	$A[7]$	$A[8]$	$A[9]$	$A[10]$	$A[11]$
14	15	16	20	21	22	25	28	30	34	45

Complessità

Complessità

- heapBuild() costa $\Theta(n)$
- maxHeapRestore() costa $\Theta(\log i)$ in un heap con i elementi
- Viene eseguita con i che varia da 2 a n

$$T(n) = \sum_{i=2}^{n} \log i + \Theta(n) = \Theta(n \log n)$$

Correttezza

Invariante di ciclo

Al passo i

- il sottovettore $A[i+1 \dots n]$ è ordinato;
- $A[1 \dots i] \leq A[i+1 \dots n]$
- $A[1]$ è la radice di un vettore heap di dimensione i.

Dimostrazione

Per esercizio

Reality check

Utilizzo (https://en.wikipedia.org/wiki/Heapsort)

Because of the $O(n \log n)$ upper bound on heapsort's running time and constant upper bound on its auxiliary storage, embedded systems with real-time constraints or systems concerned with security often use heapsort, such as the Linux kernel.

Sommario

Code con priorità

Definizione (Priority Queue)

Una coda con priorità è una struttura dati astratta, simile ad una coda, in cui ogni elemento inserito possiede una sua "priorità"

- Min-priority queue: estrazione per valori crescenti di priorità
- Max-priority queue: estrazione per valori decrescenti di priorità

Operazioni permesse

- Inserimento in coda
- Estrazione dell'elemento con priorità di valore min/max
- Modifica priorità (decremento/incremento) di un elemento inserito

Specifica

MinPriorityQueue

% Crea una coda con priorità con capacità n, vuota
PriorityQueue PriorityQueue(**int** n)

% Restituisce **true** se la coda con priorità è vuota
boolean isEmpty()

% Restituisce l'elemento minimo di una coda con priorità non vuota
Item min()

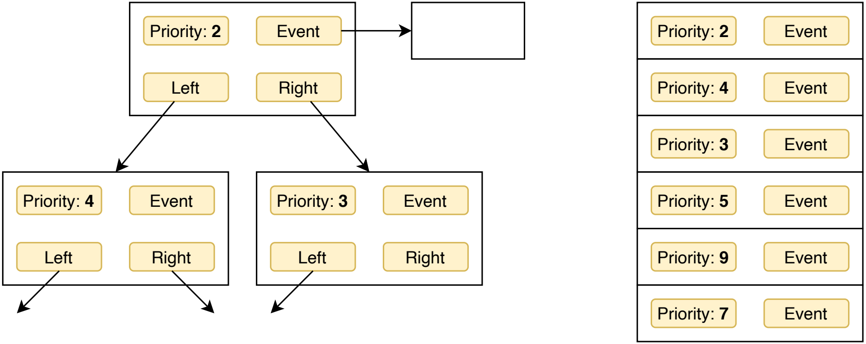
% Rimuove e restituisce il minimo da una coda con priorità non vuota
Item deleteMin()

% Inserisce l'elemento x con priorità p nella coda con priorità. Restituisce
% un oggetto PriorityItem che identifica x all'interno della coda
PriorityItem insert(Item x, **int** p)

% Diminuisce la priorità dell'oggetto identificato da y portandola a p
decrease(PriorityItem y, **int** p)

Reality Check – Simulatore event-driven

- Ad ogni evento è associato un timestamp di esecuzione
- Ogni evento può generare nuovi eventi, con timestamp arbitrari
- Una coda con min-priorità può essere utilizzata per eseguire gli eventi in ordine di timestamp



Implementazioni

Metodo	Lista/vettore non ordinato	Lista Ordinata	Vettore Ordinato	Albero RB
min()	$O(n)$	$O(1)$	$O(1)$	$O(\log n)$
deleteMin()	$O(n)$	$O(1)$	$O(n)$	$O(\log n)$
insert()	$O(1)$	$O(n)$	$O(n)$	$O(\log n)$
decrease()	$O(1)$	$O(n)$	$O(\log n)$	$O(\log n)$

Heap

Una struttura dati speciale che associa

- i vantaggi di un albero (esecuzione in tempo $O(\log n)$), e
- i vantaggi di un vettore (memorizzazione efficiente)

Implementazione con Heap

Quale versione

Implementiamo una min-priority queue, in quanto negli esempi che vedremo in seguito daremo la precedenza a elementi con priorità minore

Dettagli implementativi

- Vedremo come strutturare un vettore che memorizza coppie $\langle$ *valore*, *priorità* $\rangle$
- Vedremo come implementare minHeapRestore()
- Vedremo come implementare i singoli metodi

Memorizzazione

PriorityItem	
int *priority*	% Priorità
Item *value*	% Elemento
int *pos*	% Posizione nel vettore heap

---

swap(PriorityItem[ ] H, **int** i, **int** j)

PriorityItem $temp = H[i]$
$H[i] = H[j]$
$H[j] = temp$
$H[i].pos = i$
$H[j].pos = j$

Inizializzazione

```
PRIORITYQUEUE
int capacity                 % Numero massimo di elementi nella coda
int dim                      % Numero attuale di elementi nella coda
PRIORITYITEM[] H             % Vettore heap

PRIORITYQUEUE PriorityQueue(int n)
    PRIORITYQUEUE t = new PRIORITYQUEUE
    t.capacity = n
    t.dim = 0
    t.H = new PRIORITYITEM[1 ... n]
    return t
```

Inserimento

PRIORITYITEM $\mathsf{insert}(\text{ITEM}\ x, \textbf{int}\ p)$

precondition: $dim < capacity$

$dim = dim + 1$
$H[dim] = \textbf{new}\ \text{PRIORITYITEM}()$
$H[dim].value = x$
$H[dim].priority = p$
$H[dim].pos = dim$
$\textbf{int}\ i = dim$
$\textbf{while}\ i > 1\ \textbf{and}\ H[i].priority < H[p(i)].priority\ \textbf{do}$
    $\mathsf{swap}(H, i, p(i))$
    $i = p(i)$
$\textbf{return}\ H[i]$

minHeapRestore()

minHeapRestore(PriorityItem[] A, **int** i, **int** dim)

int $min = i$
if $l(i) \leq dim$ **and** $A[l(i)].priority < A[min].priority$ **then**
  $min = l(i)$
if $r(i) \leq dim$ **and** $A[r(i)].priority < A[min].priority$ **then**
  $min = r(i)$
if $i \neq min$ **then**
  swap(A, i, min)
  minHeapRestore(A, min, dim)

Cancellazione / lettura minimo

ITEM deleteMin()

precondition: $dim > 0$

$\mathsf{swap}(H, 1, dim)$
$dim = dim - 1$
$\mathsf{minHeapRestore}(H, 1, dim)$
return $H[dim + 1].value$

ITEM min()

precondition: $dim > 0$

return $H[1].value$

Decremento priorità

decrease(PriorityItem x, **int** p)

precondition: $p < x.priority$

$x.priority = p$
int $i = x.pos$
while $i > 1$ **and** $H[i].priority < H[p(i)].priority$ **do**
  swap$(H, i, p(i))$
  $i = p(i)$

Complessità

- Tutte le operazioni che modificano gli heap sistemano la proprietà heap
  - lungo un cammino radice-foglia (deleteMin())
  - oppure lungo un cammino nodo-radice (insert(), decrease())
- Poichè l'altezza è $\lfloor \log n \rfloor$, il costo di tali operazioni è $O(\log n)$

Operazione	**Costo**
insert()	$O(\log n)$
deleteMin()	$O(\log n)$
min()	$\Theta(1)$
decrease()	$O(\log n)$

Heap (mucchio) of presents