Heap, heap-sort, code di priorità

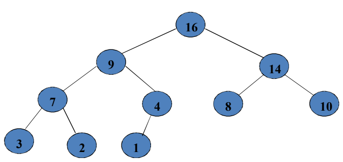
Algoritmi e strutture dati

1

Heap massimo, definizione

Def. Un albero binario a chiavi intere *H* è uno *heap massimo* se:

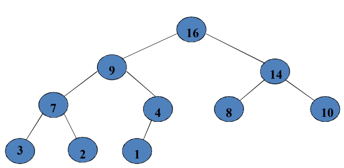
1. *H* è completo tranne al più l'ultimo livello che deve essere riempito da sinistra
2. $H.key \geq H.left.key$, $H.key \geq H.right.key$



2

Heap come array

- è conveniente rappresentare con *n* nodi con un array $H[1..n]$
- $H[1]$ sia la radice dell'albero



H =	16	9	14	7	4	8	10	3	2	1
	1	2	3	4	5	6	7	8	9	10

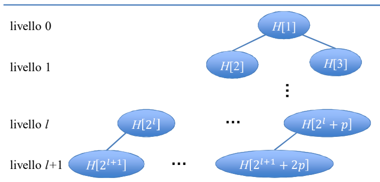
3

Left, right, parent

- per spostarci facilmente nell'array, cerchiamo la relazione fra l'indice di un nodo e l'indice del suo figlio sinistro assumendo che esista
- livelli: $l \in \{0,1,2,\dots\}$
- posizioni all'interno del livello l: $p \in \{0,1,\dots,2^l - 1\}$
- il nodo al livello l nella posizione p ha indice $2^l + p$ nell'array

4

Left, right, parent



- $H[2^{l+1} + 2p]$ è figlio sinistro di $H[2^l + p]$
- $2^{l+1} + 2p = 2(2^l + p)$
- dunque $H[2i]$ è il figlio sinistro e $H[2i+1]$ è il figlio destro di $H[i]$
- di conseguenza il padre di $H[i]$ è $H[\lfloor i/2 \rfloor]$

5

Left, right, parent

```
PARENT(H, i)
    ▷ Pre: 1 ≤ i ≤ H.N
    ▷ Post: restituisce la posizione del genitore se esiste, 0 altrimenti
return ⌊i/2⌋

LEFT(H, i)
    ▷ Pre: 1 ≤ i ≤ H.N
    ▷ Post: restituisce la posizione del figlio sinistro se esiste, i altrimenti
if 2i ≤ H.N then
   return 2i
else
   return i
end if

RIGHT(H, i)
    ▷ Pre: 1 ≤ i ≤ H.N
    ▷ Post: restituisce la posizione del figlio destro se esiste, i altrimenti
if 2i + 1 ≤ H.N then
   return 2i + 1
else
   return i
end if
```

6

Proprietà

Prop. In uno heap $H[1..n]$ le foglie occupano esattamente il semivettore $H[\lfloor n/2 \rfloor + 1..n]$

Se $n = 1$ allora $\lfloor n/2 \rfloor + 1 = 1$ ed $H[1]$ è il solo nodo.

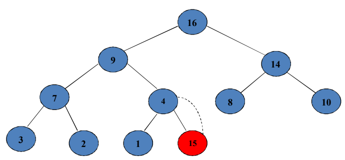
Se $n > 1$, $H[i]$ è una foglia se e solo se $n < 2i$, cioè

$$\begin{array}{rcl} n & < & 2i \\ \lfloor n/2 \rfloor & < & \lfloor i \rfloor = i \\ \lfloor n/2 \rfloor + 1 & \leq & i \end{array}$$

e $i \leq n$.

7

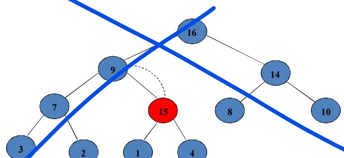
Heap massimo: inserimento (1)



HeapInsert(H,x) aggiunge *x* come foglia in *H*; quindi la fa risalire lungo il ramo cui è stato aggiunto sinché non sia ricostruito lo heap.

8

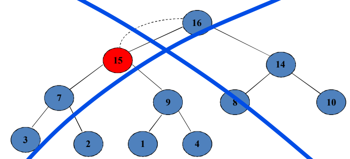
Heap massimo: inserimento (2)



HeapInsert(H,x) aggiunge *x* come foglia in *H*; quindi la fa risalire lungo il ramo cui è stato aggiunto sinché non sia ricostruito lo heap.

9

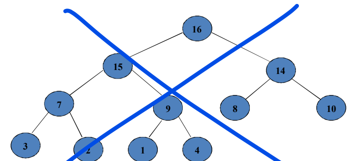
Heap massimo: inserimento (3)



HeapInsert(*H*,*x*) aggiunge *x* come foglia in *H*; quindi la fa risalire lungo il ramo cui è stato aggiunto sinché non sia ricostruito lo heap.

10

Heap massimo: inserimento (4)



HeapInsert(*H*,*x*) aggiunge *x* come foglia in *H*; quindi la fa risalire lungo il ramo cui è stato aggiunto sinché non sia ricostruito lo heap.

11

Heap massimo, inserimento

HeapInsert(*H*,*x*) aggiunge *x* come foglia in *H*; quindi la fa risalire lungo il ramo cui è stato aggiunto sinché non sia ricostruito lo heap.

$\text{HeapInsert}(H, x)$
  ▷ Pre: H è un heap
  ▷ Post: H è un heap con x inserito
$H.N \leftarrow H.N + 1$
$p \leftarrow H.N$
$H[p] \leftarrow x$
while $p > 1 \wedge H[p] > H[\text{Parent}(H, p)]$ **do**
  scambia $H[p]$ e $H[\text{Parent}(H, p)]$
  $p \leftarrow \text{Parent}(H, p)$
end while

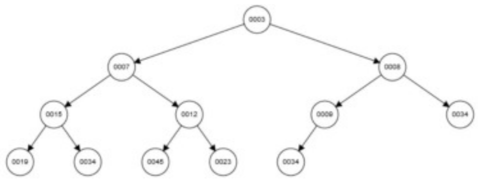
HeapInsert è $O(\log n)$

12

Heap minimo, definizione

Def. Un albero binario a chiavi intere H è uno *heap minimo* se:

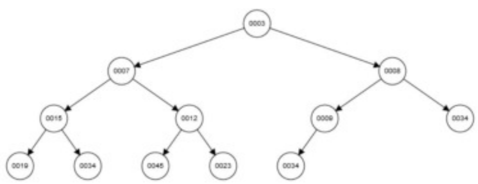
1. H è completo tranne al più l'ultimo livello che deve essere riempito da sinistra
2. $H.key \leq H.left.key$, $H.key \leq H.right.key$



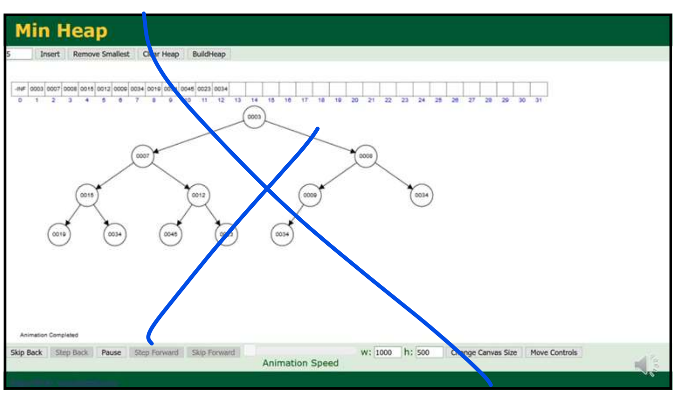
13

Heap minimo, inserimento

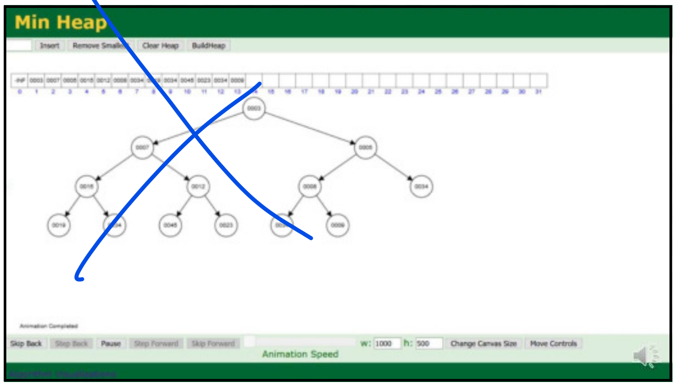
Inserimento è analogo a quello nel heap massimo ma la chiave sale se è minore della chiave del padre.
Sui lucidi successivi simuliamo l'inserimento della chiave 5.



14



15



16

Heap massimo, estrazione

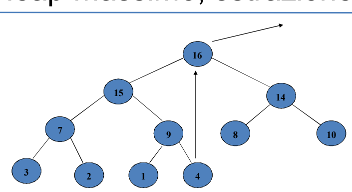
L'estrazione toglie l'elemento dalla radice. Avviene in due fasi:

1) l'elemento più a destra dell'ultimo livello rimpiazza la radice;

2) l'elemento ora in radice viene fatto discendere lungo l'albero finché non sia maggiore di entrambi i figli; nel discendere si sceglie sempre il figlio col valore massimo della chiave.

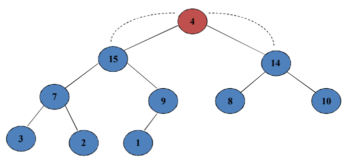
17

Heap massimo, estrazione (1)



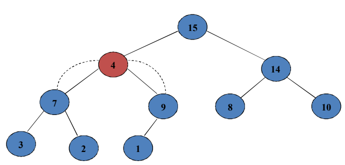
18

Heap massimo, estrazione (2)



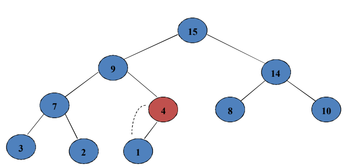
19

Heap massimo, estrazione (3)



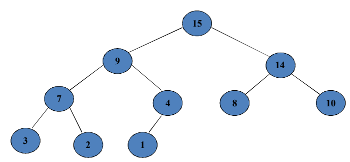
20

Heap massimo, estrazione (4)



21

Heap massimo, estrazione (5)



22

Heap massimo, estrazione

- nel momento in cui l'ultimo elemento sale nella radice, i sottoalberi con radice in *Left*(H,1) e *Right*(H,1) sono heap massimi
- la seconda fase rende heap massimo tutto l'albero
- per generalizzare, scriviamo un algoritmo che rende heap massimo l'albero che ha radice nel nodo i dato che i sottoalberi con radice in *Left*(H,i) e *Right*(H,i) sono heap massimi
- chiamiamo questo algoritmo *Heapify*(H,i)

23

Heap massimo, heapify

```
HEAPIFY(H, i)
    ▷ Pre: 1 ≤ i ≤ H.N, i sottoalberi con radice in LEFT(H, i) e RIGHT(H, i) sono heap
    ▷ Post: l'albero con radice in i è heap
m ← index of MAX{H[i], H[LEFT(H, i)], H[RIGHT(H, i)]}
if m ≠ i then
   scambia H[m] e H[i]
   HEAPIFY(H, m)
end if
```

Complessità? è $O(\log n)$.
Correttezza? si dimostra con induzione.

24

Heap massimo, estrazione

- avendo a disposizione *Heapify:*

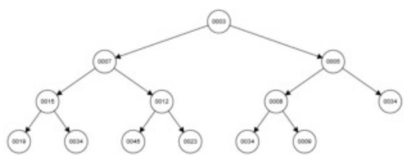
HEAPEXTRACT(H)
 ▷ Pre: H è un heap
 ▷ Post: H è un heap con etichetta massimo eliminata
$H[1] \leftarrow H[H.N]$
$H.N \leftarrow H.N - 1$
HEAPIFY($H, 1$)

Complessità? è $O(\log n)$.
Correttezza? sulla base della correttezza di Heapify.

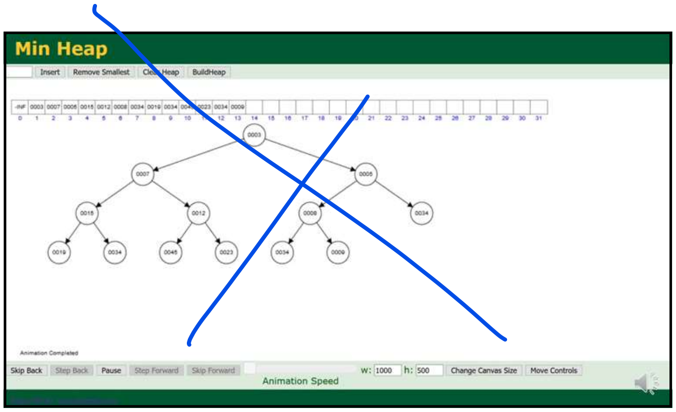
25

Heap minimo, estrazione

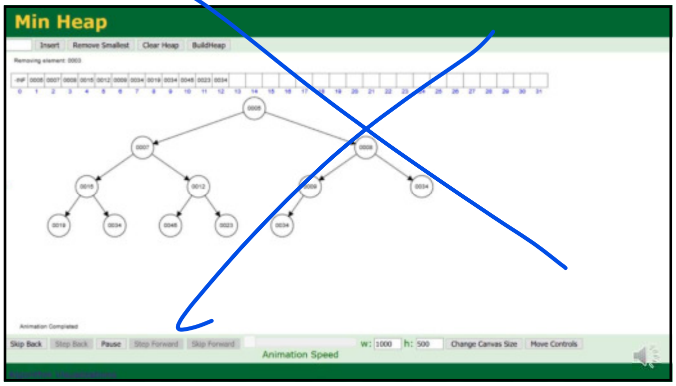
- analogo all'estrazione in un heap massimo con la differenza che si sceglie il minimo dei figli se bisogna fare un scambio
- sui lucidi simuliamo l'estrazione del minimo dal heap minimo disegnato sotto



26



27



28

~~Usi di uno heap~~

~~La struttura heap può essere impiegata per avere:~~

- ~~*code di priorità*;~~
- ~~un algoritmo di ordinamento ottimo, *Heap-Sort*.~~

29

Code di priorità (ADT)

Coda di priorità rappresenta un insieme finito S di oggetti con una funzione *Priorità*: $S \rightarrow Nat$.

```
datatype PriorityQueue, Element;
constructors:
   EmptyQueue: PriorityQueue
   Insert: PriorityQueue, Element -> PriorityQueue
   ExtractMaximum: PriorityQueue -> PriorityQueue
observations:
   Maximum: PriorityQueue -> Element
semantics:
   Insert(S,x) = S ∪ {x}
   Maximum(S) = x tale che Priorità(x) =
                 max{Priorità(y)| y ∈ S}
   ExtractMaximum(S) = S \ {Maximum(S)}
```

30

Code di priorità: implementazione

La funzione ***priorità*** si implementa codificando ogni elemento come una coppia ⟨*elemento, priorità*⟩, e strutturando lo heap in base alla seconda coordinata di ciascuna coppia (la chiave).

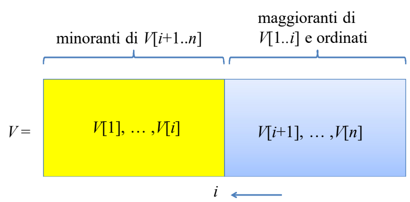
Le funzioni *Insert* e *ExtractMaximum* sono quelle viste; la funzione *Maximum* è semplicemente:

```
Maximum(H)  // pre: H è uno heap
  return H[1] // il massimo è sempre in radice
```

La funzione, non essendo distruttiva, non richiede infatti alcuna ricostruzione dello heap, ed ha complessità $O(1)$.

31

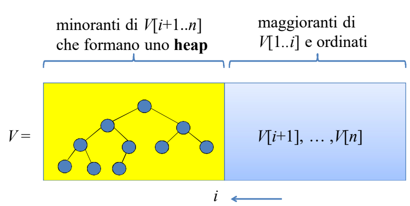
Heap-sort



Se $V[1..i]$ fosse uno heap la selezione del massimo sarebbe $O(\log i)$

32

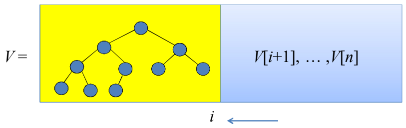
Heap-sort



Se $V[1..i]$ fosse uno heap la selezione del massimo sarebbe $O(\log i)$

33

Heap-sort

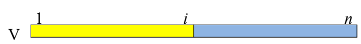


$$\begin{aligned} & \log_2 n + \log_2(n-1) + \ldots + \log_2 2 \\ = \; & \log_2(n \cdot (n-1) \cdot \ldots \cdot 2) \\ = \; & \log_2(n!) \in O(n \log n) \end{aligned}$$

34

Heap-sort

Si può sfruttare la struttura dati heap per costruire un algoritmo di ordinamento simile, per struttura, al *SelectSort* con selezione del massimo.



V[1..*i*] è uno heap   V[*i*+1..*n*] è ordinato

$\forall$ x $\in$ V[1..*i*] $\forall$ y $\in$ V[*i*+1..*n*]. x $\leq$ y

```
HeapSort(V: array)
BuildHeap(V) // riorganizza V in uno heap
for i ← size(V) downto 2 do
   scambia V[1] e V[i]
   HeapSize(V) ← HeapSize(V) - 1
   Heapify(V,1)
```

35

Heap-Sort

BuildHeap. Se $V[1..n]$ è un vettore qualsiasi, *BuildHeap*(V) lo riorganizza in modo che sia uno heap.

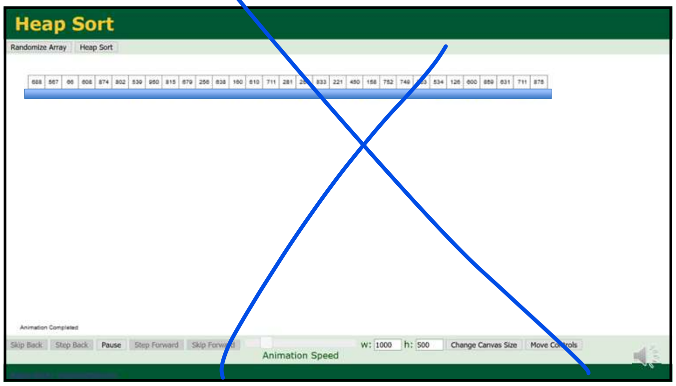
La parte $V[\lfloor n/2 \rfloor + 1 \ldots n]$ corrisponde alle foglie dell'albero, quindi gli sottoalberi che hanno come radice $V[\lfloor n/2 \rfloor + 1], V[\lfloor n/2 \rfloor + 2] \ldots, V[n-1], V[n]$ sono trivialmente heap già all'inizio.

Possiamo iterare *Heapify* da $\lfloor n/2 \rfloor$ a 1 :

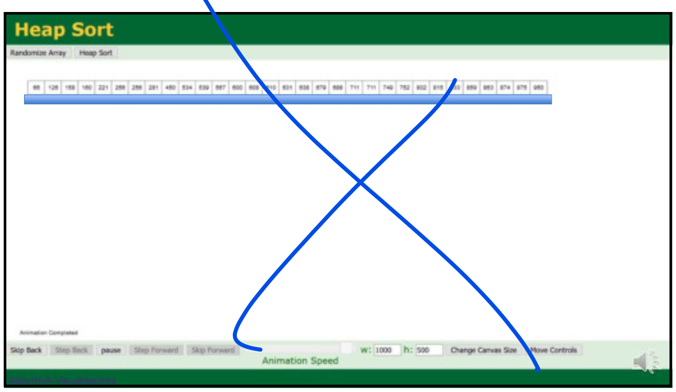
```
BuildHeap(V: array)
for i ← ⌊length(V)/2⌋ downto 1 do
   Heapify(V,i)
```

Nota: un confine superiore alla complessità di *BuildHeap* è $O(n \log n)$ (si itera una procedura $O(\log n)$). (Questa stima grossolana si può raffinare sino a mostrare che in realtà *BuildHeap* è lineare.)

36



37



38

Heap-sort

A differenza del *Selection-Sort*, che è $O(n^2)$, *HeapSort* ha complessità $O(n \log n)$, quindi è un algoritmo ottimo per l'ordinamento. Ciò si deve all'efficienza della selezione del massimo nel semivettore sinistro, che ha complessità logaritmica:

```
HeapSort(V: array)
BuildHeap(V) // O(n log n) (o meglio O(n))
for i ← Length(V) downto 2 do // O(n) cicli
   scambia V[1] e V[i]
   HeapSize(V) ← HeapSize(V) - 1
   Heapify(V,1) // O(log n)
```

Allora: $O(n) + O(n)\, O(\log n) = O(n) + O(n \log n) = O(n \log n)$.

39