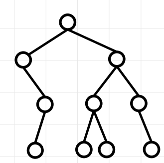
Heap

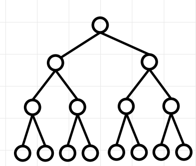
1

Alberi binari

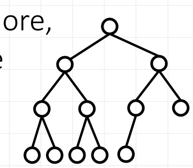
Albero binario:
Ogni nodo ha zero, uno, o due successori (ordinati)



Albero binario completo:
Tutte le foglie hanno la stessa profondità e tutte le profondità sono completamente piene



Albero binario *quasi* completo:
Albero binario completo tranne che per il livello inferiore, che è pieno da sinistra verso destra solo parzialmente



2

Alberi binari quasi completi

Si consideri un albero binario quasi completo si altezza h

Se il livello inferiore contiene 1 elemento:

num. di elementi $n = (1 + 2 + ... + 2^{h-1}) + 1 = (2^h - 1) + 1 = 2^h$

Se il livello inferiore è pieno:

numero di elementi $n = 1 + 2 + ... + 2^h = 2^{h+1} - 1$

Quindi in ogni caso: $2^h \leq n \leq 2^{h+1} - 1 < 2^{h+1}$

$\rightarrow h \leq log\ n < h + 1$

$\rightarrow log\ n - 1 < h \leq log\ n$

$\rightarrow h = \lfloor log\ n \rfloor$

Sarà importante per heap sort

Heap: definizione formale

Una Heap è un albero binario quasi completo.

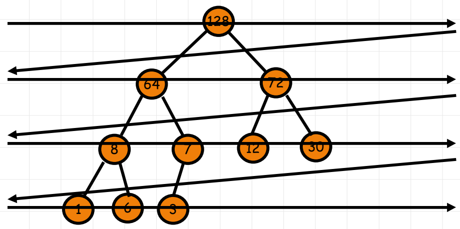
Quasi significa che possono mancare alcune foglie consecutive a partire dall'ultima foglia di destra.

Per ogni nodo *i*: $value(i) \leq value(parent(i))$

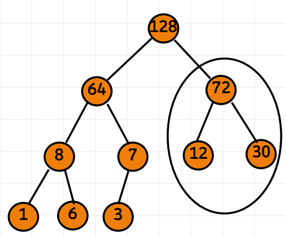
Nota 1: il massimo si trova nella radice (max-heap)

Nota 2: non c'è nessuna relazione tra il valore di un nodo e quello di un suo fratello

heap in un vettore



Heap binarie




```
A={128, 64, 72, 8, 7, 12, 30, 1, 6,  3}
    1   2   3  4  5   6   7  8  9  10
```

A(6) = 12

heap in un vettore

Radice ➡ posizione 1

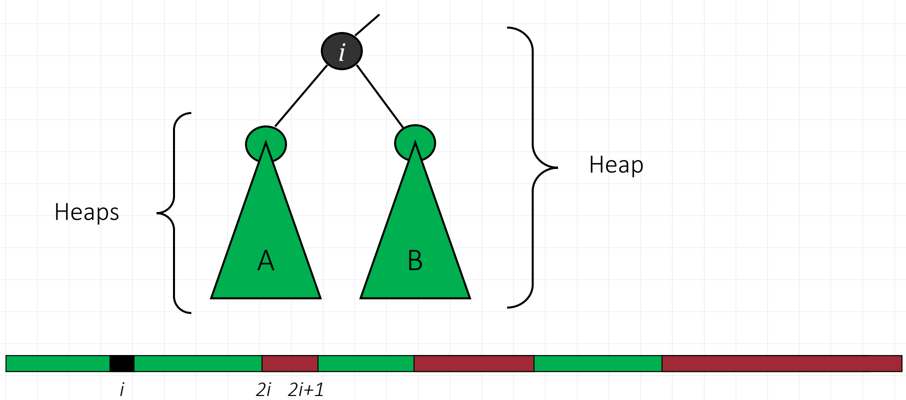
Per ogni nodo in posizione *i*:

left-child(i) ➡ posizione *2i*

right-child(i) ➡ posizione *2i+1*

$parent(i) = \lfloor i/2 \rfloor$

Aggiunta di un elemento: heapify

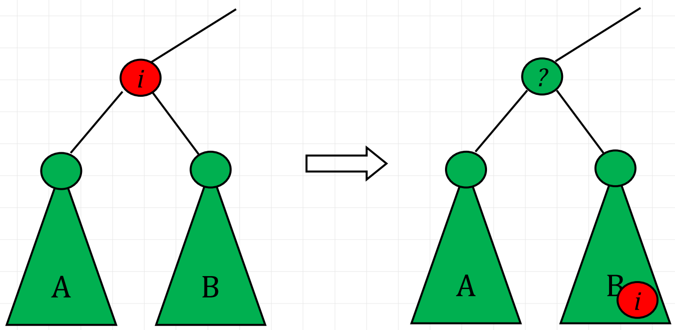


parte del vettore già heapizzato

elemento da aggiungere alla sotto heap (verde)

Heapify

IDEA: facciamo scendere
il nodo *i* nell'albero fino
a trovare la sua posizione.



Vittorio Maniezzo - Universita di Bologna 9

9

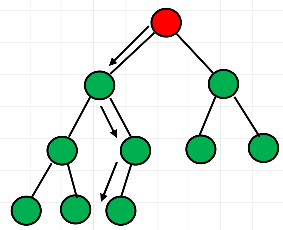
Heapify(A,i)

```
HEAPIFY(A,i)
    l = left(i)
    r = right(i)
    if (l ≤ heap-size(A) and A[l]>A[i])
       then largest=l
       else largest=i
    if (r ≤ heap-size(A) and A[r]>A[largest])
       then largest=r
    if (largest ≠ i)
       then SWAP(A[i],A[largest])
            HEAPIFY(A,largest)
```

Vittorio Maniezzo - Universita di Bologna 10

10

Heapify: costo computazionale



Caso pessimo: il nodo si sposta fino ad arrivare alle foglie.

Heapify impiega tempo costante ad ogni livello per sistemare *A[i], A[left(i)] e A[right(i)].*

Esegue aggiustamenti locali al massimo ***height(i)*** volte dove ***height(i) = O(log(n))***

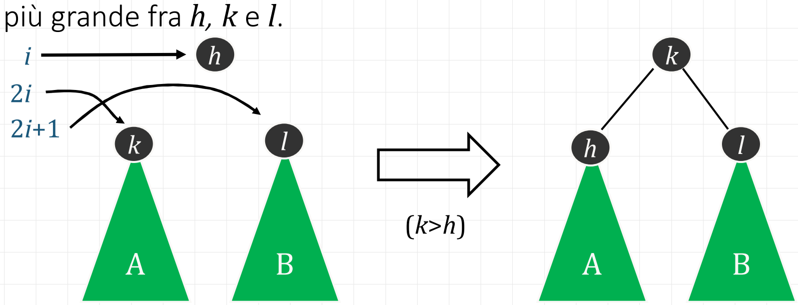
Build-heap

Idea: si trasforma in una heap un albero binario con radice in posizione i.

I dati sono contenuti in un array a, con $a[i] = h$

Si assume che gli alberi binari con radice in $2i$ e $2i + 1$ siano già delle heap. $a[2i]=k$ e $a[2i+1]=l$

Se necessario si fa uno scambio per garantire che in posizione i ci sia il più grande fra h, k e l.



Build-heap

```
BUILD-HEAP(A)
    heap-size(A)=length(A)
    for i=⌊length(A)/2⌋ downto 1
    do HEAPIFY(A,i)
```

Gli elementi in posizione da ⌊`length(A)/2`⌋ a `length(A)` sono già heap unitarie.

Analisi approssimativa:

- ogni chiamata a heapify costa $O(log(n))$.
- chiamiamo heapify $O(n)$ volte,
- quindi build-heap = $O(n\ log(n))$

Domanda (esercizio): build-heap = $\Theta(n\ log(n))$?

Priority Queue (Code a Prioritá)

Dati:

un insieme di elementi, ognuno dei quali ha una chiave (un intero per esempio).

Operazioni:

- inserimento,
- trova il massimo,
- estrazione del massimo (massima chiave).

Applicazioni delle PQ:
Job scheduling, event-driven simulations, …

Implementazione (facile) usando vettori

Prima soluzione: vettore ordinato.

Ricerca massimo: $\Theta(1)$ operazioni
estrazione massimo: $\Theta(1)$ operazioni
inserimento: $\Theta(n)$ operazioni

Seconda soluzione vettore non ordinato.

Ricerca massimo: $\Theta(n)$ operazioni
estrazione massimo: $\Theta(n)$ operazioni
inserimento: $\Theta(1)$ operazioni

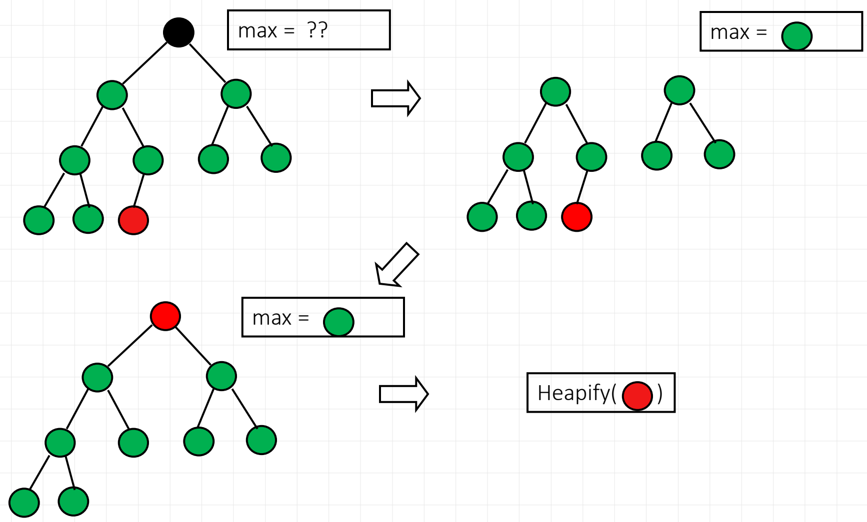
Si può fare meglio ???

PQ implementate con Heap

```
EXTRACT-MAX(A)
    max=A[1]
    A[1]=A[heapsize(A)]
    heapsize(A)=heapsize(A)-1
    HEAPIFY(A,1)
    return max
```

$$O(log(n))$$

PQ implementate con Heap



PQ implementate con Heap

```
INSERT(A,x)
    heapsize(A)=heapsize(A)+1
    i=heapsize(A)
    while (i>1 and A[parent(i)]<x)
        do A[i]=A[parent(i)]
           i=parent(i)
    A[i]=x
```

$O(log(n))$

Heap Sort: l'idea.

Per ordinare in senso crescente.

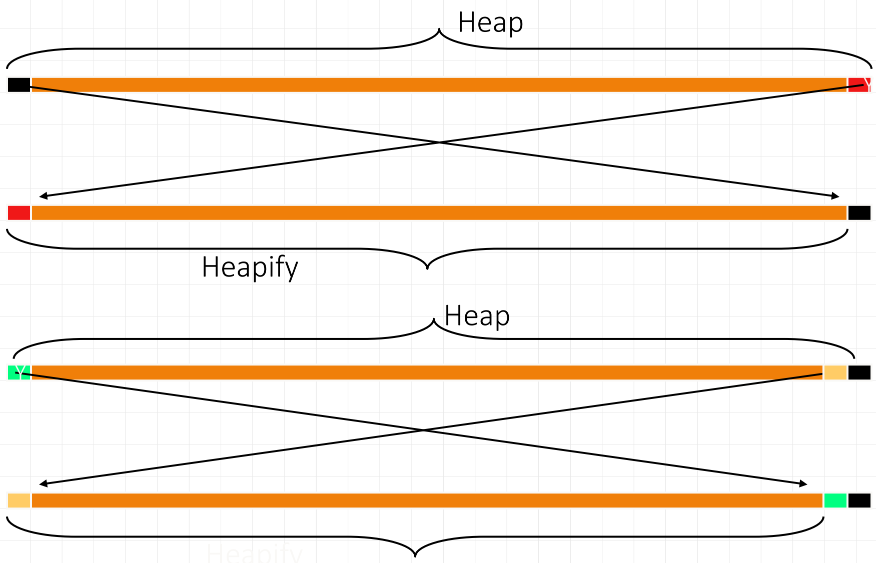
Prima parte:

- si trasforma l'array in input in una *max*-heap

Seconda parte:

- si scambia il dato nella radice con il dato dell'ultimo nodo
- si esclude l'ultimo nodo dalla heap (non lo si tocca più)
- si ricostruisce la heap

Heap Sort: l'idea.



Heap Sort

```
HEAPSORT(A)

BUILDHEAP(A)
for i=length(A) downto 2
   do EXCHANGE(A[1],A[i])
      heap-size[A] = heap-size(A)-1
      HEAPIFY(A,1)
```

Oppure (meglio)

```
HEAPSORT(A)
BUILDHEAP(A)
for i=length(A) downto 2
  A[i] = EXTRACTMAX();
```

$O(n \log(n))$

Heap sort, complessitá

Complessità nel caso pessimo di heapsort.

- buildHeap ripete O(n) volte il ciclo interno
- Si fanno n − 1 chiamate a Heapify
- Ogni chiamata costa O(log n)
- Quindi heapSort ∈ O(n · log n)

Domanda: heapsort è in-place?

Heapsort, esempio *(da wikipedia)*

Dati:	6, 5, 3, 1, 8, 7, 2, 4	
Heap:	8, 6, 7, 4, 5, 3, 2, 1	
Heap:	7, 6, 3, 4, 5, 1, 2	8
Heap:	6, 5, 3, 4, 2, 1	7, 8
Heap:	5, 4, 3, 1, 2	6, 7, 8
Heap:	4, 2, 3, 1	5, 6, 7, 8
Heap:	3, 2, 1	4, 5, 6, 7, 8
Heap:	2, 1	3, 4, 5, 6, 7, 8
Heap:	1	2, 3, 4, 5, 6, 7, 8
Heap:		1, 2, 3, 4, 5, 6, 7, 8

Ordinamenti, proprietá

	Insertion sort	Merge sort	Heap sort	Quick sort
Caso pessimo	n^2	$n \log(n)$	$n \log(n)$	n^2
Caso medio	n^2	$n \log(n)$	$n \log(n)$	$n \log(n)$
Caso ottimo	n	$n \log(n)$	$n \log(n)$	$n \log(n)$

= in place