Max-Heap

Idea

Struttura dati per accedere **sempre al valore maggiore** in maniera più efficiente rispetto ad un vettore ordinato.

Spesso viene usato per salvare una **lista di chiavi** per accedere a dati satellite, difatti useremo questo nome per intendere i valori al suo interno

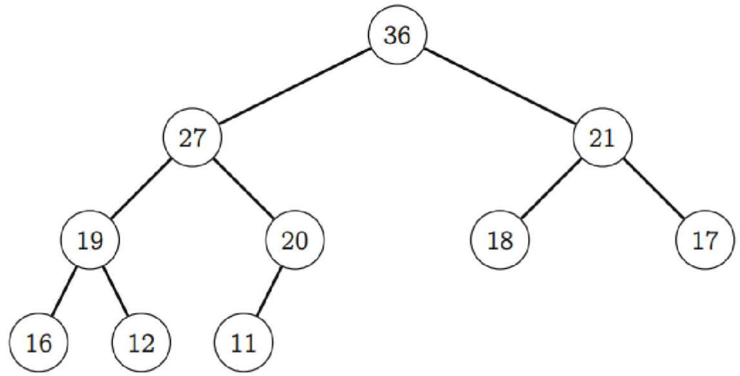
Un min-heap sarà **analogo con i minimi** e con un operazione di **decremento**

Rappresentazione ad Albero

Proprietà: Albero

Nonostante il max-heap sia un **vettore**, possiamo rappresentarlo come un **albero binario** dove

- Ogni nodo padre ha una chiave **maggiore o uguale dei singoli figli**, avendo quindi la **chiave massima nella radice**
- Tutti i piani meno l'ultimo sono **completamente bilanciati**
- Le **foglie devono essere consecutive** da sinistra verso destra



1	2	3	4	5	6	7	8	9	10
36	27	21	19	20	18	17	16	12	11

Per calcolare l'altezza dell'albero lo consideriamo completo, considerando che se mancassero dei valori basterà arrotondare alla **parte intera superiore**

$$n = 2^h - 1 \iff n + 1 = 2^h \iff h = \lceil \log_2(n+1) \rceil = \Theta(\log(n))$$

Operazioni e Scopo

Operazioni

- [Costruire un Max Heap]
- [Trovare il massimo]
- [Estrarre il massimo]
- [Inserire un valore]
- [Incrementare un valore]

In generale, le operazioni che hanno un costo pesante in un vettore ordinato (dinamiche) hanno un costo leggero in uno disordinato e viceversa.

Essendo un heap una via di mezzo, le operazioni possibili vengono fatte quasi tutte con un costo medio di $\log(n)$, portando per le operazioni possibili quindi una buonissima efficienza senza rinunciare ad una delle due parti

Esempio: Code con Priorità

In una coda con priorità, le operazioni possibili sono

1. **Trovare** il massimo
2. **Estrarre** il massimo
3. **Inserire** un valore

Se implementiamo con un **vettore ordinato**, otteniamo come costi

1. $\Theta(1)$
2. $\Theta(1)$
3. $\Theta(n)$

Se implementiamo con un **vettore disordinato**, otteniamo come costi

1. $\Theta(n)$
2. $\Theta(n)$
3. $\Theta(1)$

Possiamo notare come le operazioni hanno esattamente i costi invertiti.

Ma se implementiamo con un max-heap, otterremo un costo non sempre costante ma sempre efficiente

1. $\Theta(1)$
2. $\Theta(\log(n))$
3. $\Theta(\log(n))$

Navigazione nel Vettore

Spostamento tra i nodi dell'albero

Idea

Traduciamo gli spostamenti sull'albero in spostamento dell'indice sul vettore, ignorando la fuoriuscita da quest'ultimo

PARENT(i)

```
return floor(i / 2)
```

Parametri: *i=indice del nodo nel vettore*
Return: *indice dell'ipotetico nodo padre*

LEFT(i)

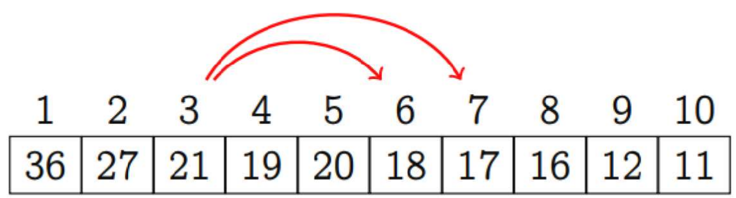
```
return 2i
```

Parametri: *i=indice del nodo nel vettore*
Return: *indice dell'ipotetico figlio sinistro*

RIGHT(i)

```
return 2i + 1
```

Parametri: *i=indice del nodo nel vettore*
Return: *indice dell'ipotetico figlio destro*



Costo Computazionale

$\Theta(1)$

Heapify

Trasformazione di un singolo nodo in un heap parziale

Idea

Preso un nodo

- Se è maggiore dei singoli figli o è una foglia è al suo posto
- Se no scambialo di posto con il maggiore tra i due figli e ripeti ricorsivamente

MAX-HEAPIFY(A, i)

```
l = LEFT(i)
r = RIGHT(i)

if l <= A.heap_size and A[l] > A[i]
        largest = l
else
        largest = i
if r <= A.heap_size and A[r] > A[largest]
        largest = r

if largest != i
        exchange A[i] with A[largest]
        MAX-HEAPIFY(A, largest)
```

***Parametri**: A=heap, i=indice del nodo di partenza*

Il caso peggiore si presenterà quando il nodo deve essere spostato lungo tutta l'altezza dell'albero

Costo Computazionale

$$\Theta(\log n)$$

Build Heap

Trasformazione completa in heap del vettore

Idea

Applico Heapify su ogni nodo.
Possiamo saltare tutte le foglie in quanto senza figli, partendo quindi dalla metà del vettore risalendo fino alla cima

BUILD-MAX-HEAP(A)

```
A.heap_size = A.length
for i = floor(A.length / 2) downto 1
        MAX-HEAPIFY(A, i)
```

***Parametri**: A=heap*

Sembrerebbe che il costo computazionale sia $\Theta(n\log(n))$, ma consideriamo l'**equazione della funzione partendo dalla radice**

$$\begin{aligned} T(n) = \log(n) + \\ & + 2(\log(n) - 1) \\ & + 4(\log(n) - 2) \\ & + \dots \\ & + \frac{n}{2}(1) \end{aligned}$$

$$\sum_{i=0}^{\log(n)-1} 2^i(\log(n) - i)$$

Ma se consideriamo invece l'**equazione partendo dal fondo**

$$T(n) = \frac{n}{2}(1) + \\ + \frac{n}{4}(2) \\ + \dots \\ + \log(n)$$

$$\sum_{i=1}^{\log(n)} \frac{n}{2^i}(i)$$

Ma sapendo che i è una costante e che

$$\sum_{i=1}^{\infty} \frac{1}{2^i} = k > 0$$

Otteniamo

$$\sum_{i=1}^{\log(n)} \frac{n}{2^i}(i) = kn = \Theta(n)$$

Costo Computazionale

$\Theta(n)$

Ricerca

Idea

Se l'heap è ben costruito, il valore massimo/minimo corrisponderà alla radice dell'albero

HEAP-MAXIMUM(A)

```
return A[1]
```

Parametri: *A=heap*
Return: *valore massimo nello heap*

Costo Computazionale

$\Theta(1)$

Modifica

Operazioni di **incremento e decremento (modifica)** di una chiave

Idea

Dopo la modifica del valore di un nodo, bisognerà assicurare il mantenimento della struttura dell'heap

HEAP-INCREASE-KEY(A, i, key)

```
if key < A[i]
        error new key is smaller than current one

A[i] = key
while i > 1 and A[PARENT(i)] < A[i]
        exchange A[i] with A[PARENT(i)]
        i = PARENT(i)
```

Parametri: *A=heap, i=indice chiave da modificare, key=nuovo valore*

Il caso peggiore si presenterà quando il nodo deve essere spostato lungo tutta l'altezza dell'albero

Costo Computazionale

$\Theta(\log(n))$

Inserimento

Inserimento di una nuova chiave

> 💡 **Idea**
>
> La chiave verrà inserita consecutivamente all'ultima foglia, per poi essere posizionata correttamente nella struttura

MAX-HEAP-INSERT(A, key)

```
A.heap_size = A.heap_size + 1
A[A.heap_size] = -infinity
HEAP-INCREASE-KEY(A, A.heap_size, key)
```

Parametri: *A=heap, key=chiave da inserire*

Presenterà lo stesso costo computazionale dell'incremento

> ⓘ **Costo Computazionale**
>
> $\Theta(\log(n))$

Estrazione

> 💡 **Idea**
>
> Dopo aver trovato il massimo, bisogna rimuoverlo e ricostruire l'heap. Per farlo, possiamo sostituire la radice estratta con l'ultimo valore del vettore, e poi chiamare Heapify per sistemarlo.

HEAP-EXTRACT-MAX(A)

```
if A.heap_size < 1
        error heap underflow

max = HEAP-MAXIMUM(A)
A[1] = A[A.heap_size]
A.heap_size = A.heap_size - 1
MAX-HEAPIFY(A, 1)

return max
```

Parametri: *A=heap*
Return: *valore massimo dello heap, estraendolo*

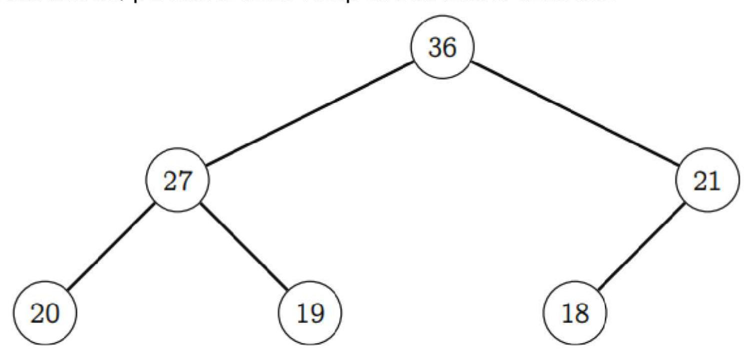
Presenterà lo stesso costo computazionale di Heapify

> ⓘ **Costo Computazionale**
>
> $\Theta(\log(n))$

Heap Sort

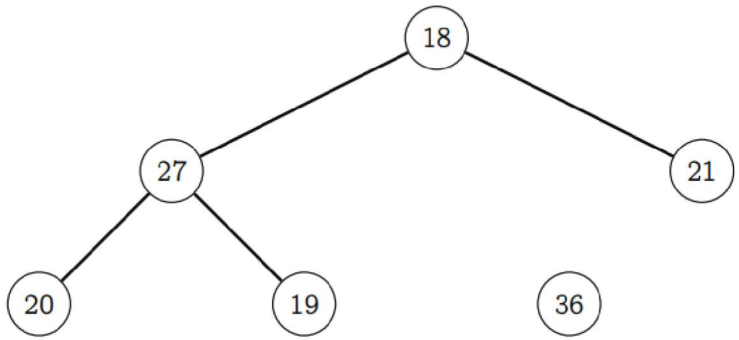
> 💡 **Idea**
>
> Siccome sappiamo che in un max heap la radice è sempre il valore massimo, possiamo iterativamente estrarre la radice e posizionarla come ultimo elemento del vettore, fino ad ordinarlo

Per ogni iterazione, partiamo dallo heap che abbiamo costruito



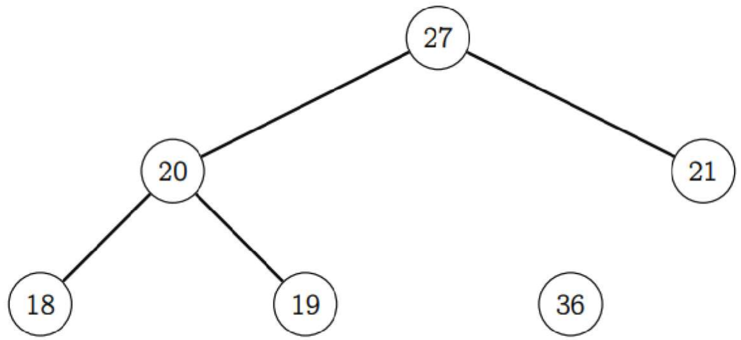
$$\underbrace{36 \mid 27 \mid 21 \mid 20 \mid 19 \mid 18}_{\text{heap size}}$$

Estraiamo il massimo e mettiamo l'ultimo valore come nuova radice



$$\underbrace{18 \mid 27 \mid 21 \mid 20 \mid 19}_{\text{heap size}} \mid 36$$

Risistemiamo il vettore per essere un heap così da estrarre nuovamente il massimo al prossimo ciclo



$$\underbrace{27 \mid 20 \mid 21 \mid 18 \mid 19}_{\text{heap size}} \mid 36$$

Possiamo riutilizzare le [operazioni già viste]() per gli heap

HEAP-SORT(A)

```
BUILD-MAX-HEAP(A) // da qui in avanti A sarà un heap

for i = A.length downto 2
        exchange A[1] with A[i]
        A.heap_size = A.heap_size - 1
        MAX-HEAPIFY(A, 1)
```

Parametri: *A=vettore*

Costo Computazionale

Andando a calcolare il costo computazionale, possiamo notare

- [BUILD-MAX-HEAP]() con costo $\Theta(n)$
- [MAX-HEAPIFY]() con costo $\Theta(\log(n))$ ripetuto $n-1$ volte

> ⓘ **Costo Computazionale**
>
> $$\Theta(n\log(n))$$

Algoritmo In-Place

Il vantaggio di heap-sort rispetto ad altri algoritmi di ordinamento è quello di essere un **algoritmo in-place**

> **Definizione**
>
> Definiamo in-place un algoritmo che lavora direttamente sulle strutture in input **senza allocare strutture dati d'appoggio**

Questo è utile quando si lavora con **sistemi in cui anche la memoria è una risorsa altamente limitata** e che necessita di una gestione efficiente