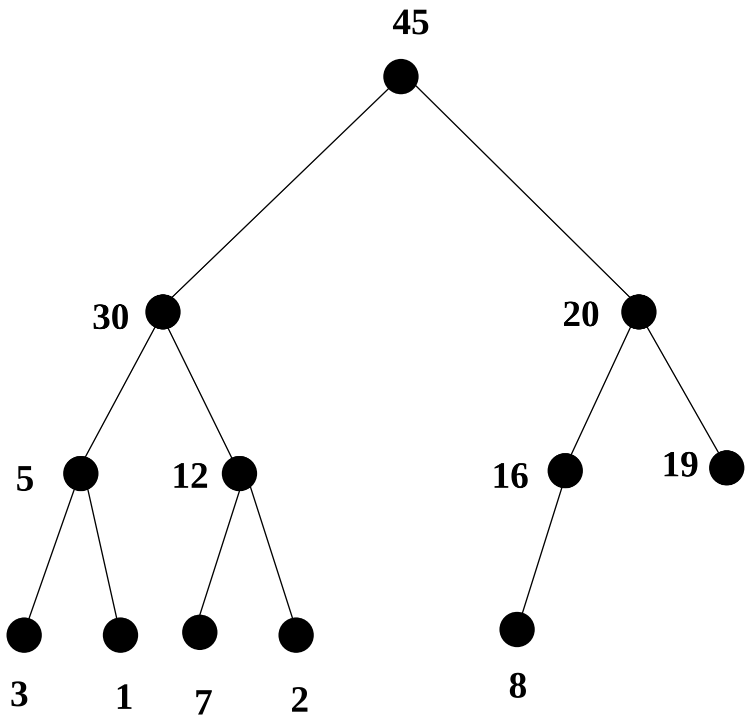
Algoritmi e Strutture di Dati I

Massimo Franceschet
http://www.sci.unich.it/~francesc

Heap

La struttura di dati **heap** rappresenta un insieme dinamico mediante un **albero binario quasi completo** le cui chiavi soddisfano la seguente **proprietà degli heap**:

> *la chiave di ogni nodo è maggiore o uguale della chiave dei propri figli.*


45
30
20
5
12
16
19
3
1
7
2
8

Operazioni sugli heap

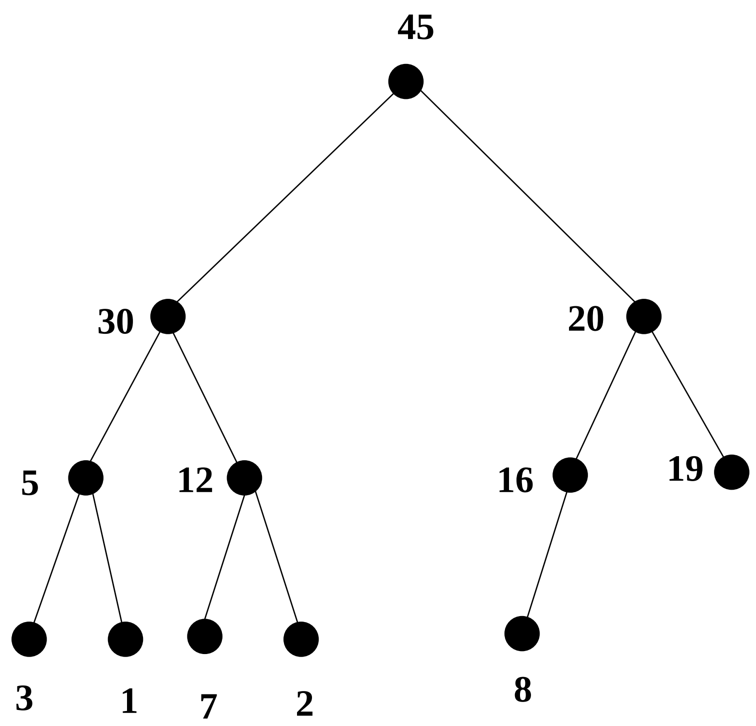
Uno heap possiede un attributo $heapsize(A)$ che contiene il numero degli elementi inseriti nello heap A, e le seguenti operazioni:

- $HeapEmpty(A)$, che verifica se lo heap A è vuoto;
- $HeapFull(A)$, che verifica se lo heap A è pieno;
- $HeapBuild(A)$, che costruisce uno heap a partiere da un vettore A qualsiasi;
- $HeapMax(A)$ che ritorna l'elemento massimo dello heap A;

- $HeapInsert(A, x)$ che inserisce nello heap A un nuovo elemento x;
- $HeapDelete(A, i)$ che rimuove dallo heap A l'elemento con indice i;
- $HeapModofyKey(A, i, k)$ che modifica la chiave dell'elemento con indice i nello heap A facendogli assumere il valore di k.

Rappresentazione di uno heap

Implementiamo uno heap usando un vettore A e inserendo in A i nodi dello heap per livelli, dall'alto al basso, da sinistra a destra.



	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16
A	45	30	20	5	12	16	19	3	1	7	2	8				

heapSize(A) = 12

Dato un nodo in posizione i:

- l'indice del nodo padre è $\lfloor i/2 \rfloor$. Se $\lfloor i/2 \rfloor < 1$, cioè se $i = 1$, allora i non ha padre;
- l'indice del figlio sinistro è $2i$. Se $2i > heapsize(A)$, allora i non ha figlio sinistro;
- l'indice del figlio destro è $2i + 1$. Se $2i + 1 > heapsize(A)$, allora i non ha figlio destro.

Parent(i)

1: **return** i *div* 2

Left(i)

1: **return** $2i$

Right(i)

1: **return** $2i + 1$

Sia $n = heapsize(A)$. Valgono le sequenti proprietà:

- i nodi interni dello heap hanno indice da 1 a $\lfloor n/2 \rfloor$, e le foglie dello heap hanno indice da $\lfloor n/2 \rfloor + 1$ fino a n. Dunque in uno heap ci sono $\lfloor n/2 \rfloor$ nodi interni e $\lceil n/2 \rceil$ foglie;
- il minimo numero di nodi di uno heap di altezza h è 2^h. Il massimo numero di nodi di uno heap di altezza h è $2^{h+1} - 1$. Dunque $n = \Theta(2^h)$ e $h = \Theta(\log n)$;
- i nodi ad altezza h in uno heap sono al massimo $\lceil n/2^{h+1} \rceil$.

Heapify

La procedura $Heapify(A, i)$ assume che i sottoalberi sinistro e destro del nodo i siano degli heap, ma $A[i]$ può essere più piccolo dei proprio figli.

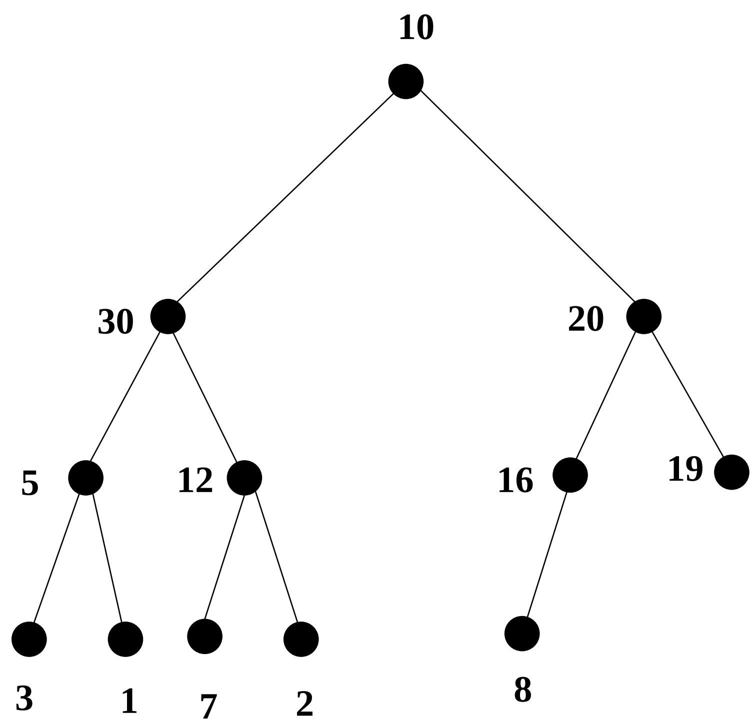
Se questo è il caso, la procedura fa scivolare l'elemento $A[i]$ lungo un cammino dell'albero in modo da ristabilire la proprietà degli heap.

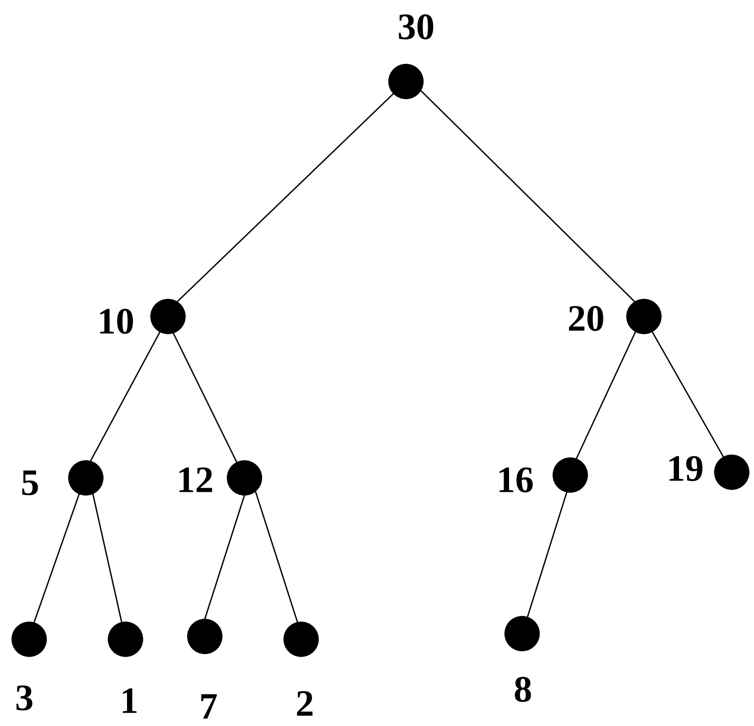
Heapify(A,i)

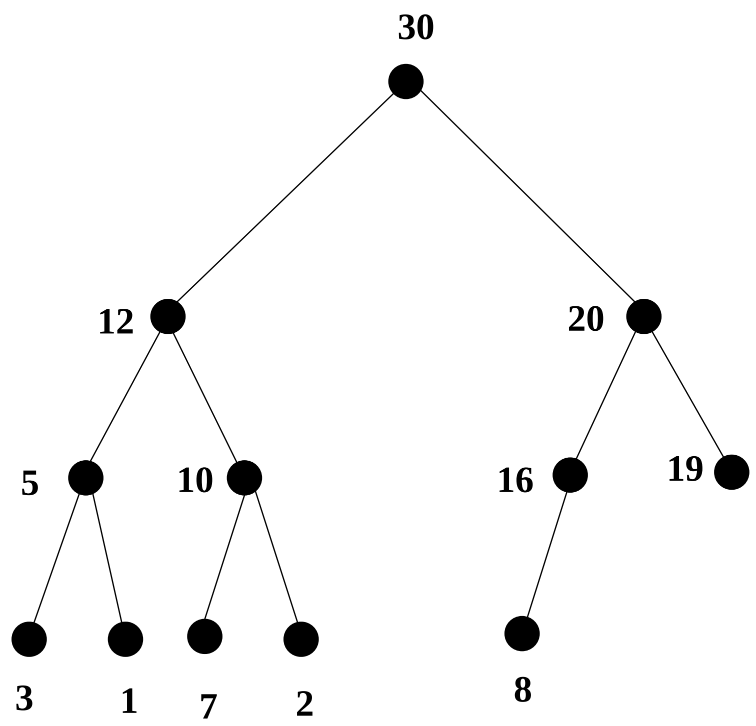
$l \leftarrow Left(i)$
$r \leftarrow Right(i)$
$largest \leftarrow i$
if $(l \leq heapsize(A))$ **and** $(A[l] > A[i])$ **then**
    $largest \leftarrow l$
end if
if $(r \leq heapsize(A))$ **and** $(A[r] \geq A[largest])$ **then**
    $largest \leftarrow r$
end if
if $largest \neq i$ **then**
    $Exchange(A, i, largest)$
    $Heapify(A, largest)$
end if

La complessità di $Heapify(A, i)$ è $\Theta(h)$, con h l'altezza dell'albero con radice i.

L'altezza di uno heap è logaritmica nel numero di nodi. Dunque la complessità di $Heapify(A, i)$ è $\Theta(\log n)$, ove n è il numero di nodi dell'albero radicato in i.


10
30
20
5
12
16
19
3
1
7
2
8


30
10
20
5
12
16
19
3
1
7
2
8


30
12
20
5
10
16
19
3
1
7
2
8

HeapEmpty(A)

if $heapsize(A) = 0$ **then**
  return TRUE
else
  return FALSE
end if

HeapFull(A)

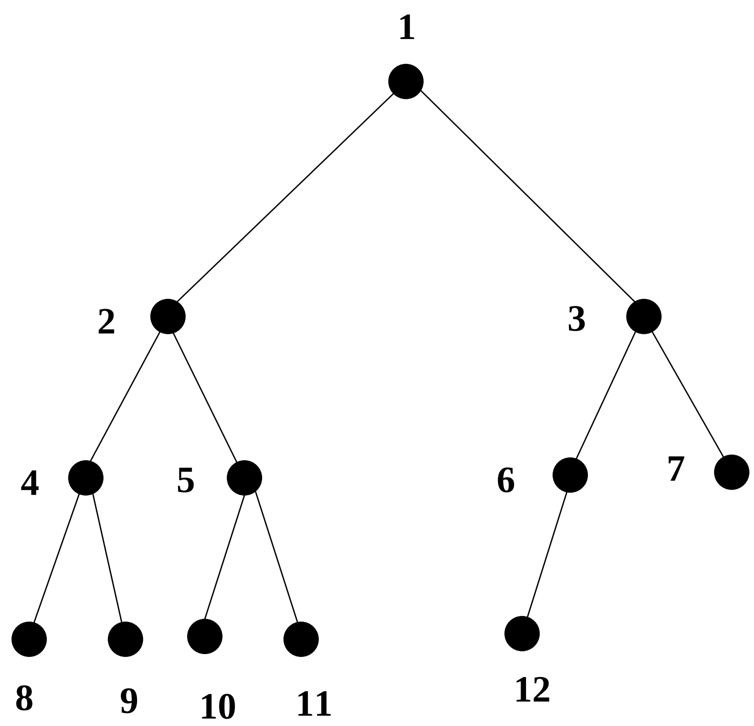
if $heapsize(A) = length(A)$ **then**
  return TRUE
else
  return FALSE
end if

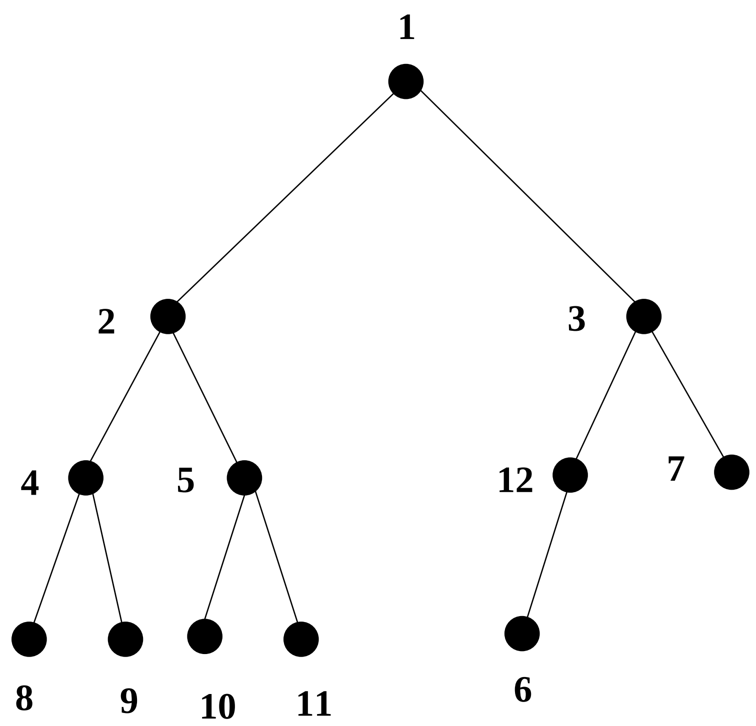
HeapMax(A)

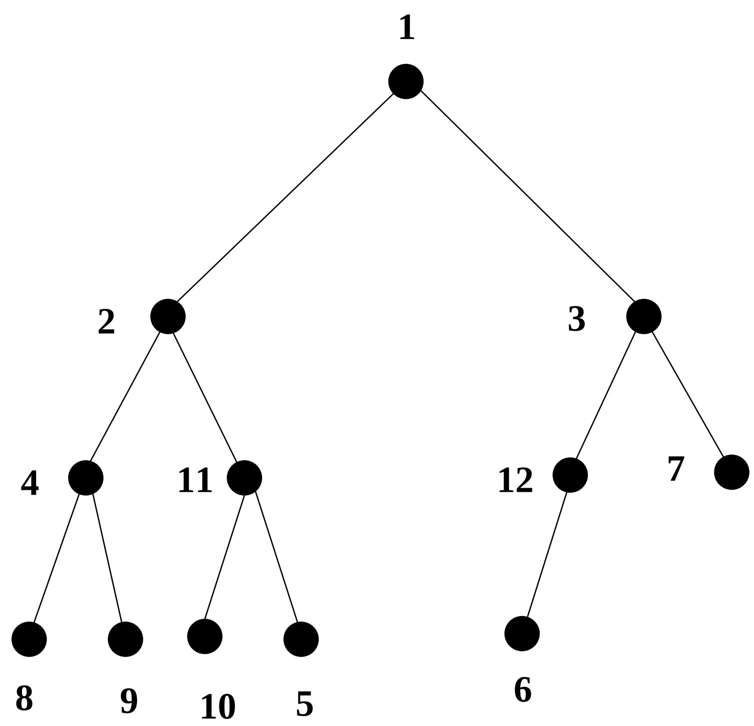
if $HeapEmpty(A)$ **then**
    return NIL
end if
return $A[1]$

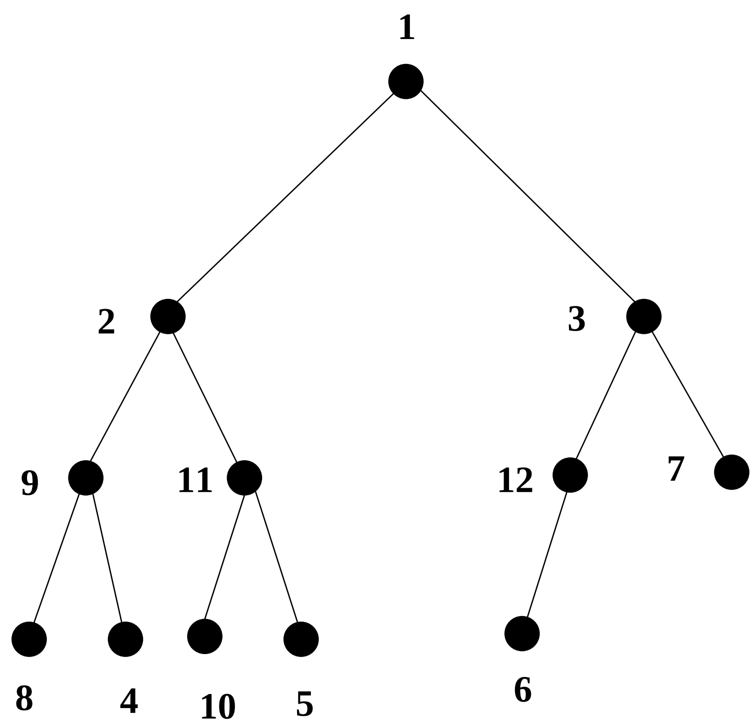
HeapBuild(A)

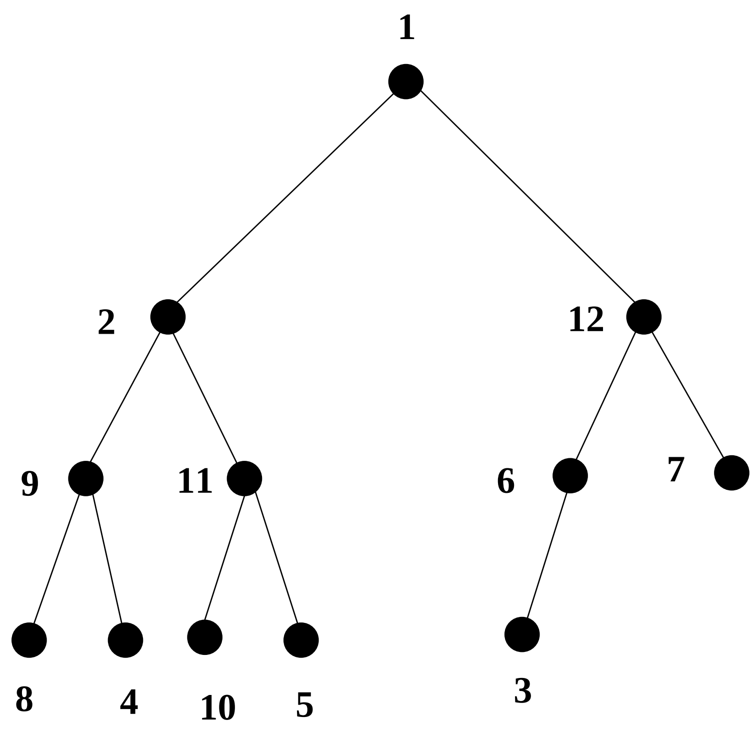
$heapsize(A) \leftarrow length(A)$
for $i \leftarrow heapsize(A)$ div 2 **downto** 1 **do**
    $Heapify(A, i)$
end for

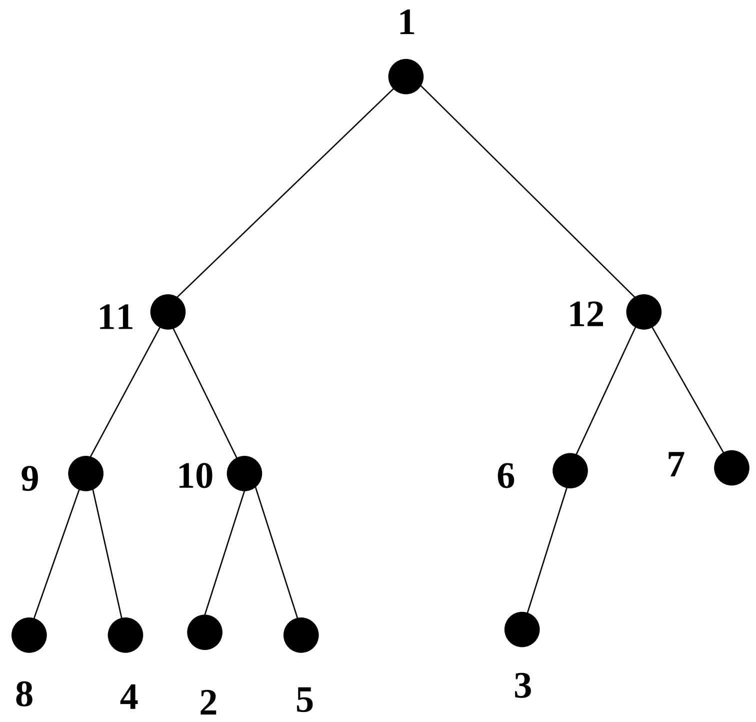

1
2
3
4
5
6
7
8
9
10
11
12

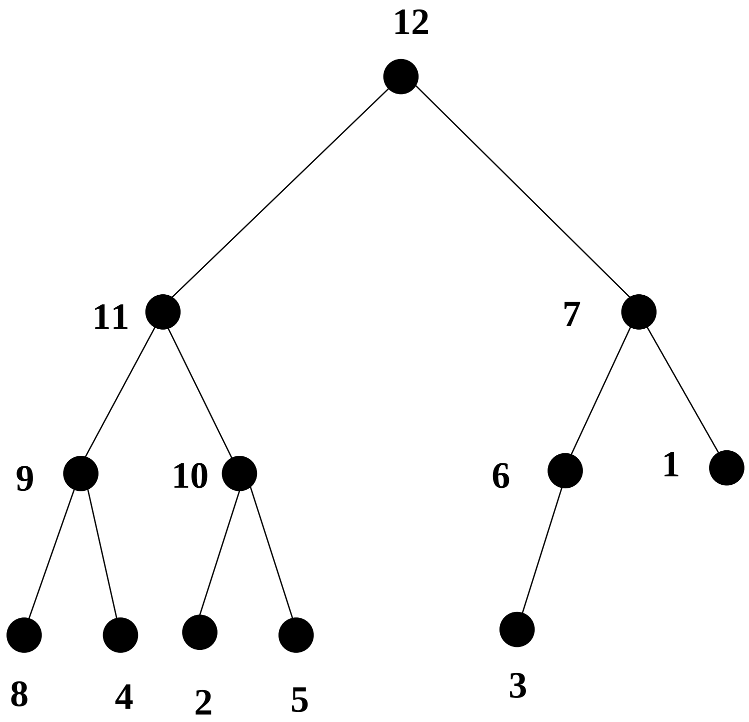

1
2
3
4
5
12
7
8
9
10
11
6


1
2
3
4
11
12
7
8
9
10
5
6


1
2
3
9
11
12
7
8
4
10
5
6


1
2
12
9
11
6
7
8
4
10
5
3


1
11
12
9
10
6
7
8
4
2
5
3


12
11
7
9
10
6
1
8
4
2
5
3

Complessità di HeapBuild

HeapBuild(A)

```
heapsize(A) ← length(A)
for i ← heapsize(A) div 2 downto 1 do
    Heapify(A, i)
end for
```

Sia $n = heapSize(A)$. Il ciclo for viene ripetuto $\lfloor n/2 \rfloor$ volte. Il costo della chiamata $Heapify(A, i)$ all'interno del ciclo è, nel caso pessimo, pari all'altezza del nodo i.

Dato uno heap di n nodi, la sua altezza è $\lfloor \log n \rfloor$ e il numero di nodi di altezza h è al più $\lceil n/2^{h+1} \rceil$.

Sommando le complessità di $Heapify$, livello per livello, a partire dalle foglie fino alla radice, abbiamo:

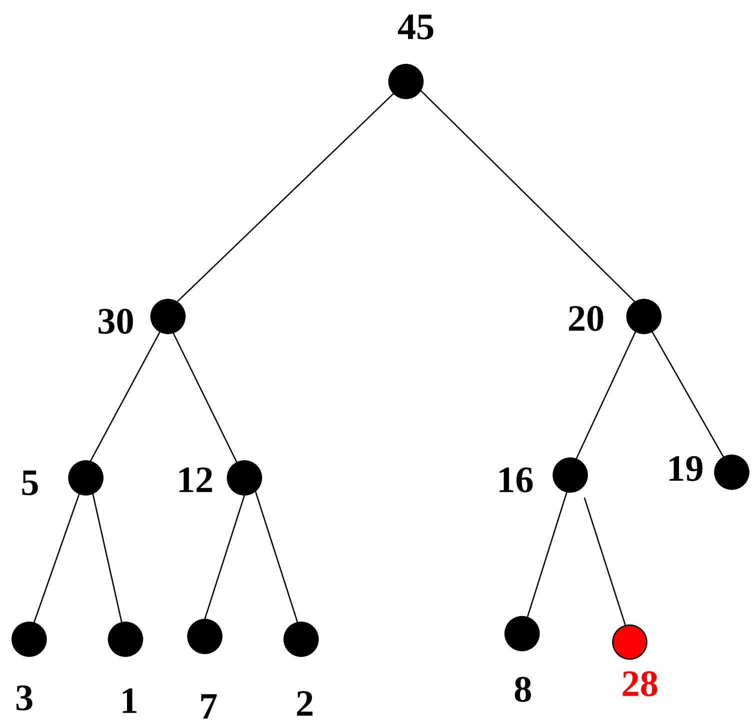
$$\sum_{h=0}^{\lfloor \log n \rfloor} \lceil n/2^{h+1} \rceil \Theta(h) = \Theta(n \sum_{h=0}^{\lfloor \log n \rfloor} h/2^h)$$

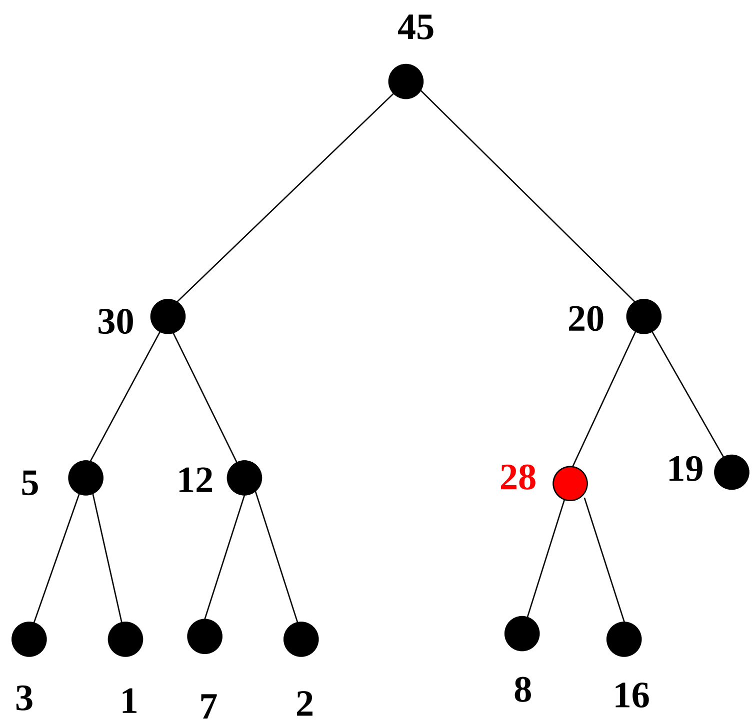
Dato che $\sum_{h=0}^{\infty} h/2^h = 2$, abbiamo che $0 \leq \sum_{h=0}^{\lfloor \log n \rfloor} h/2^h < 2$ e quindi

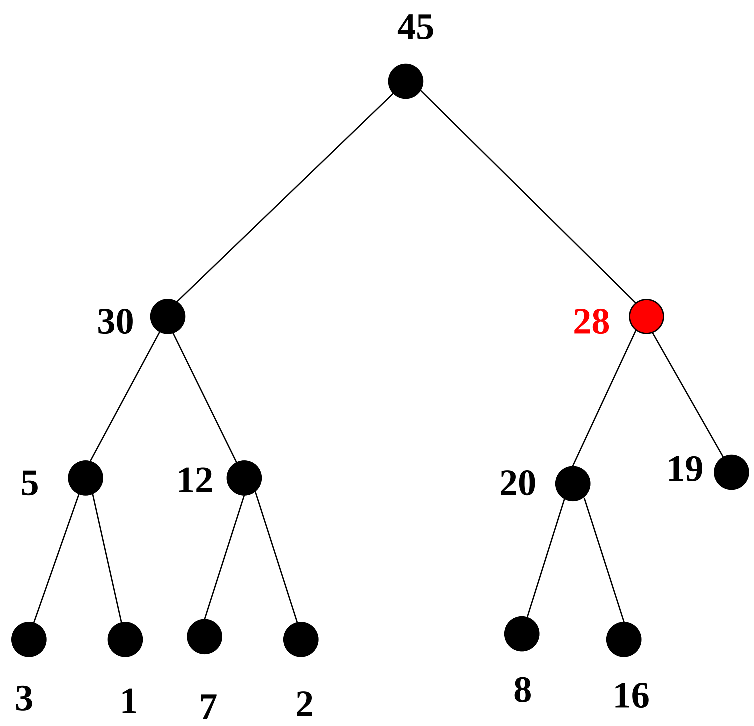
$$\Theta(n \sum_{h=0}^{\lfloor \log n \rfloor} h/2^h) = \Theta(n)$$

HeapInsert(A,x)

if $HeapFull(A)$ **then**
    error OVERFLOW
end if
$heapsize(A) \leftarrow heapsize(A) + 1$
$A[heapsize(A)] \leftarrow x$
$i \leftarrow heapsize(A)$
while $i > 1$ **and** $A[Parent(i)] < A[i]$ **do**
    $Exchange(A, i, Parent(i))$
    $i \leftarrow Parent(i)$
end while

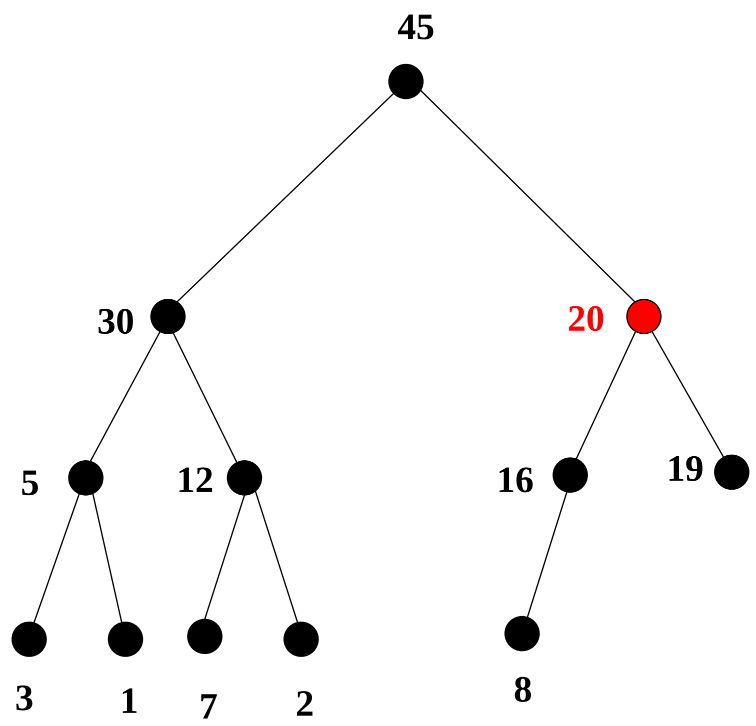

45
30
20
5
12
16
19
3
1
7
2
8
28

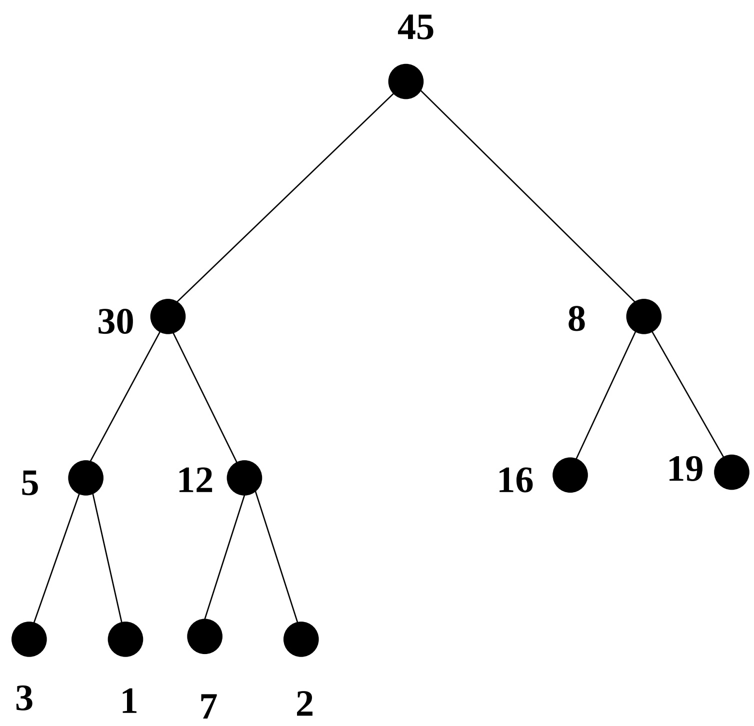

45
30
20
5
12
28
19
3
1
7
2
8
16

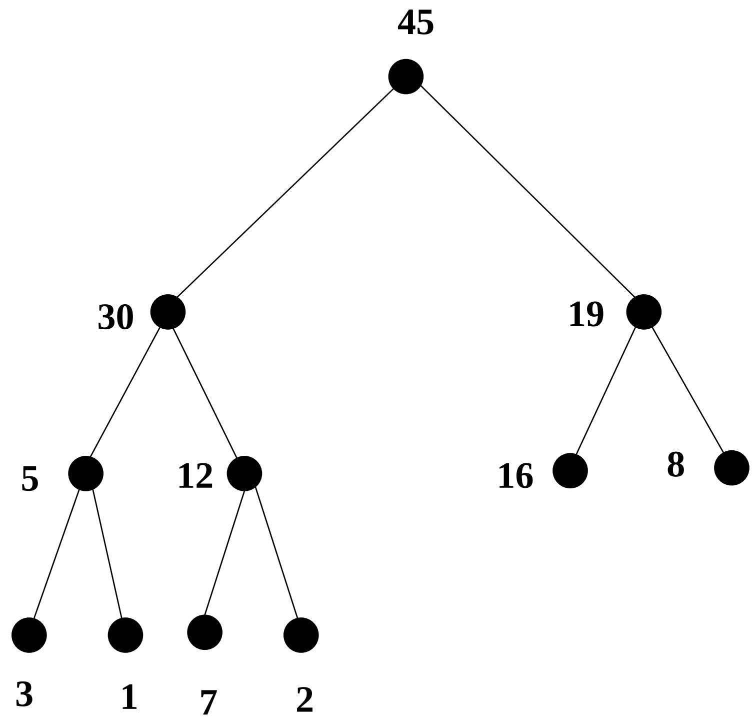

45
30
28
5
12
20
19
3
1
7
2
8
16

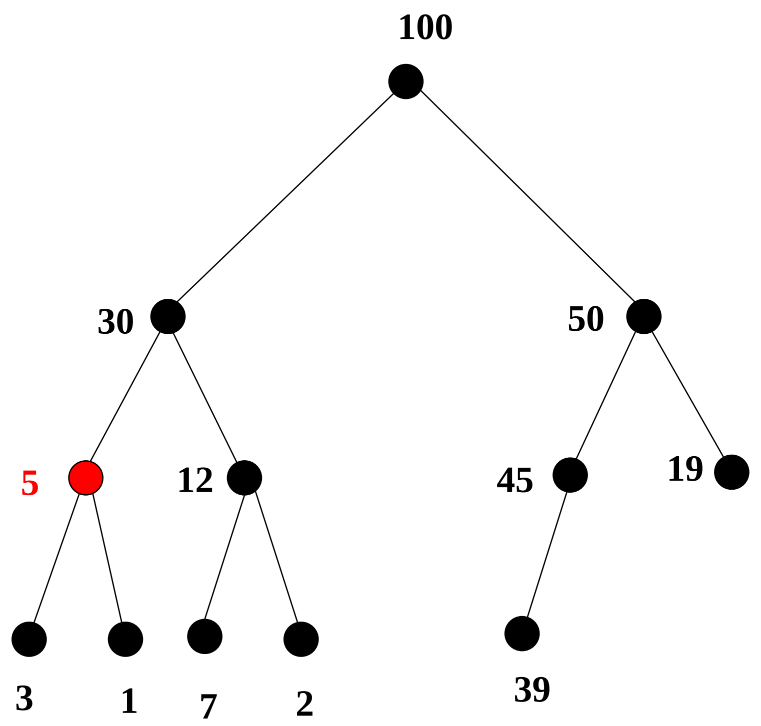
HeapDelete(A,i)

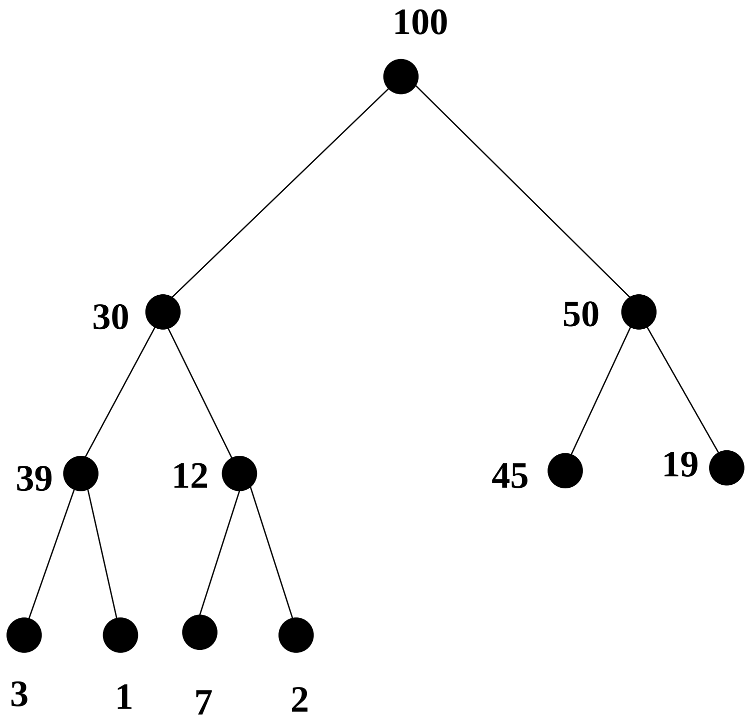
if $HeapEmpty(A)$ **then**
    error UNDERFLOW
end if
$old \leftarrow A[i]$
$new \leftarrow A[heapsize(A)]$
$A[i] \leftarrow new$
$heapsize(A) \leftarrow heapsize(A) - 1$
if $new < old$ **then**
    $Heapify(A, i)$
else
    while $(i > 1)$ **and** $(A[Parent(i)] < A[i])$ **do**
        $Exchange(A, i, Parent(i))$
        $i \leftarrow Parent(i)$
    end while
end if

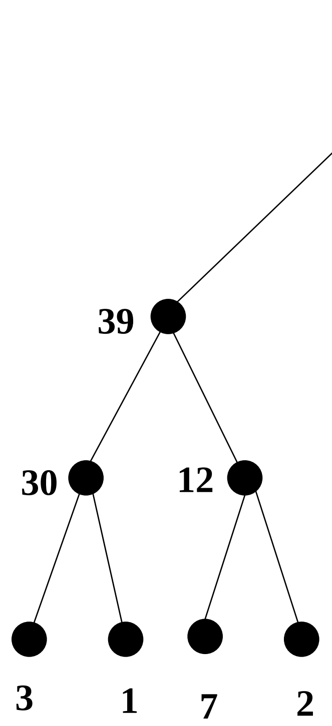

45
30
20
5
12
16
19
3
1
7
2
8


45
30
8
5
12
16
19
3
1
7
2


45
30
19
5
12
16
8
3
1
7
2


100
30
50
5
12
45
19
3
1
7
2
39


100
30
50
39
12
45
19
3
1
7
2


100
39
50
30
12
45
19
3
1
7
2

HeapModifyKey(A,i,k)

$old \leftarrow A[i]$
$new \leftarrow k$
$A[i] \leftarrow new$
if $new < old$ **then**
    $Heapify(A, i)$
else
    while $(i > 1)$ **and** $(A[Parent(i)] < A[i])$ **do**
        $Exchange(A, i, Parent(i))$
        $i \leftarrow Parent(i)$
    end while
end if

La complessità dell'inserimento, della cancellazione e della modifica di chiave è $\Theta(h) = \Theta(\log n)$, ove h è l'altezza dello heap e n il numero di nodi dello heap.

Esercizio *Implementare le procedure* $HeapInsert$ *e* $HeapDelete$ *usando* $ModifyKey$.

HeapInsert(A,x)

if $HeapFull(A)$ **then**
  error OVERFLOW
end if
$heapsize(A) \leftarrow heapsize(A) + 1$
$A[heapsize(A)] \leftarrow -\infty$
$HeapModifyKey(A, heapsize(A), x)$

HeapDelete(A,i)

if $HeapEmpty(A)$ **then**

    error UNDERFLOW

end if

$new \leftarrow A[heapsize(A)]$

$heapsize(A) \leftarrow heapsize(A) - 1$

$HeapModifyKey(A, i, new)$

Esercizio *Scrivere* $HeapMin(A)$ *che ritorna l'indice del minimo dello hep* A*.*

HeapMin(A)

1: **return** $ArrayMin(A, (heapsize(A)\ div\ 2) + 1, heapsize(A))$

La complessità è $\Theta(n/2) = \Theta(n)$, ove $n = heapsize(A)$.

Esercizio *Scrivere* $HeapSearch(A, i, k)$ *che ritorna l'indice dell'elemento con chiave* k *cercando nel sottoalbero di* A *radicato in* i*, oppure* NIL *se tale elemento non esiste.*

HeapSearch(A,i,k)

```
if (i > heapsize(A)) or (A[i] < k) then
    return NIL
end if
if A[i] = k then
    return i
end if
l ← HeapSearch(A, Left(i), k)
if l ≠ NIL then
    return l
else
    return HeapSearch(A, Right(i), k)
end if
```

La complessità è $\Theta(n)$, ove n è il numero di nodi dell'albero radicato in i.

Code con priorità

Una **coda con priorità** (priority queue) è una coda i cui oggetti hanno assegnata una priorità.

Gli oggetti vengono inseriti in coda con una priorità assegnata. L'elemento cancellato dalla coda non è il primo ad essere entrato, ma quello con priorità **massima**. Inoltre, la priorità di un oggetto può essere modificata.

Implementiamo una coda con priorità usando uno heap.

PriorityInsert(A,x)

1: $HeapInsert(A, x)$

ExtractMax(A)

1: $max \leftarrow A[1]$
2: $HeapDelete(A, 1)$
3: **return** max

ModifyPriority(A,i,k)

1: $HeapModifyKey(A, i, k)$

Queste operazioni hanno complessità logaritmica nel numero di oggetti presenti in coda.

min-heap e max-heap

Un **max-heap** è uno heap che soddisfa la proprietà:

> *la chiave di ogni nodo è maggiore o uguale della chiave dei propri figli.*

Un **min-heap** è uno heap che soddisfa la proprietà:

> *la chiave di ogni nodo è minore o uguale della chiave dei propri figli.*

Quindi un max-heap ha come radice il massimo, e un min-heap ha come radice il minimo.

Una coda con priorità **basate sul massimo** è implementata da un **max-heap**. Dunque l'elemento estratto è quello con **priorità più alta**.

Una coda con priorità **basate sul minimo** è implementata da un **min-heap**. Dunque l'elemento estratto è quello con **priorità più bassa**.