Algoritmi e Strutture Dati

HeapSort II

Complessità di Heapify

$T(n) = max(O(1),max(O(1),T(?) + O(1))$

```
H
    l = SINISTRO(i)
    r = DESTRO(i)
    IF l ≤ heapsize[A] AND A[l] > A[i]
        THEN maggiore = l
        ELSE maggiore = i
    IF r ≤ heapsize[A] AND A[r] > A[maggiore]
        THEN maggiore = r
    IF maggiore ≠ i
        THEN "scambia A[i] e A[maggiore]"
            Heapify(A,maggiore)
```

$O(1) =$ { (lines from l = SINISTRO(i) to THEN maggiore = r)

IF maggiore ≠ i } $= O(1)$

$O(1) =$ { THEN "scambia A[i] e A[maggiore]"

$T(?) =$ { Heapify(A,maggiore)

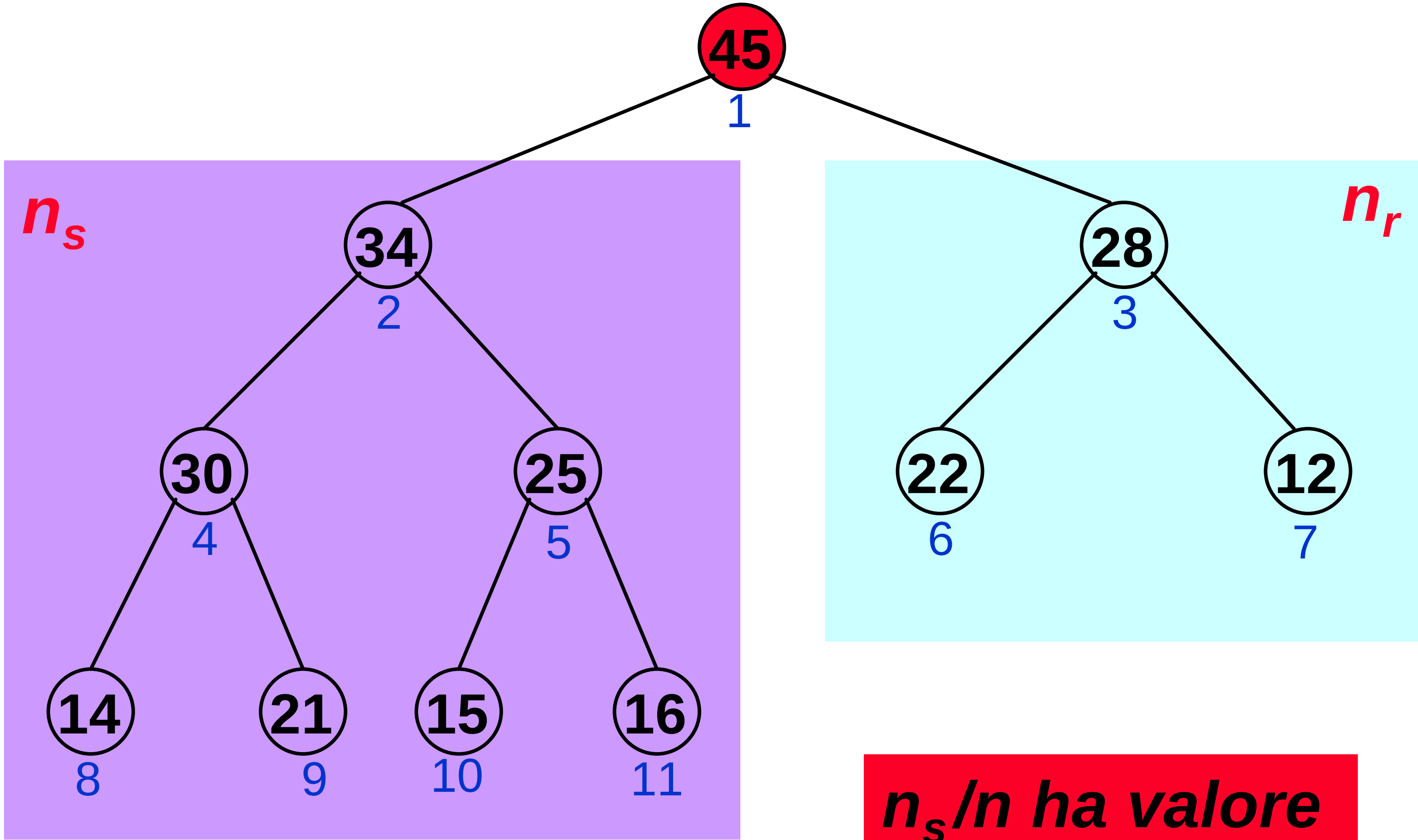
Complessità di Heapify:caso peggiore

$$T(n) = max(O(1),max(O(1),T(?) + O(1))$$

Nel *caso peggiore* *Heapify* ad ogni chiamata ricorsiva, viene eseguito su un numero di nodi che è minore dei 2/3 del numero di nodi correnti n.

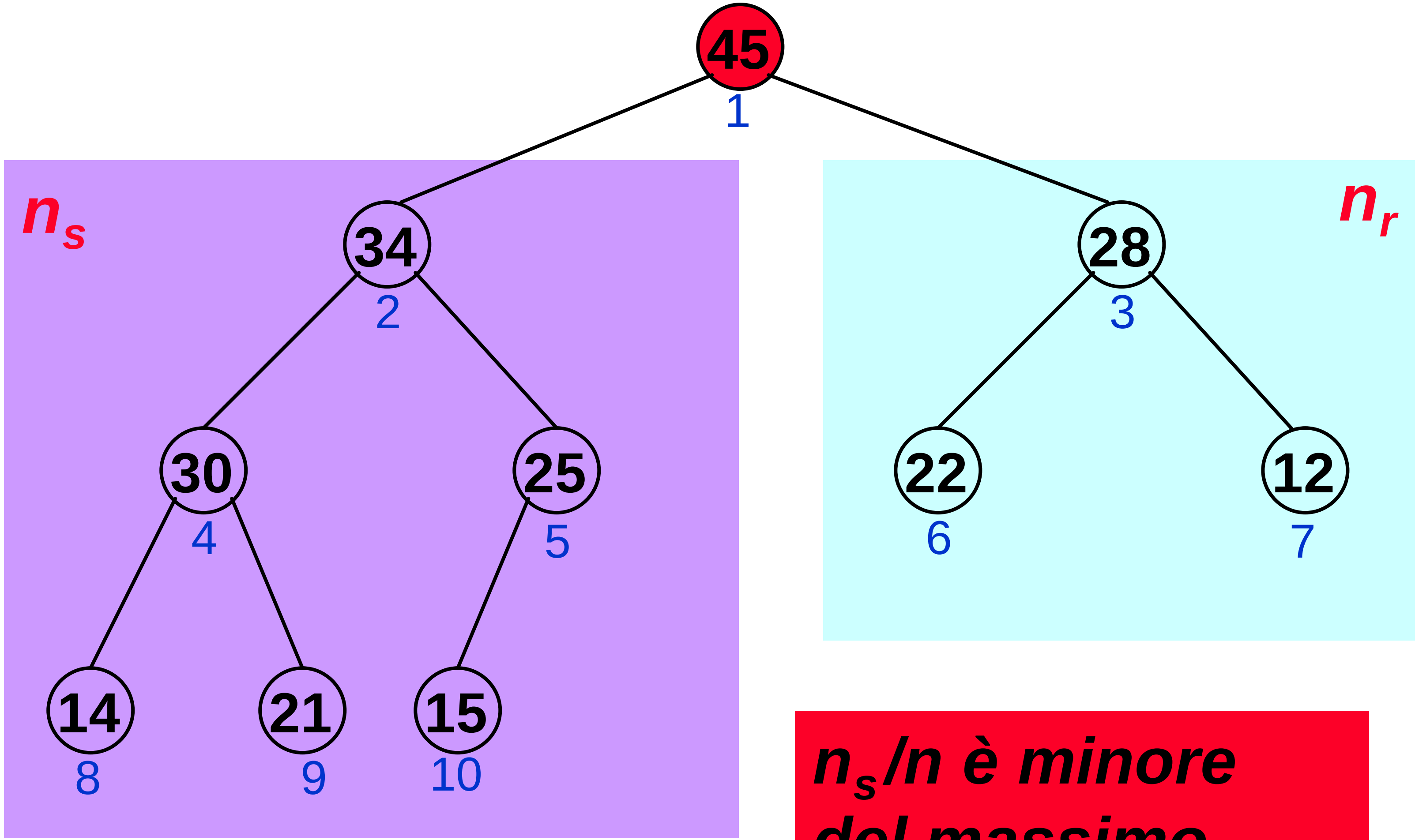
Cioè il numero di nodi n_s del sottoalbero su cui *Heapify* è chiamato ricorsivamente è al più $2/3\,n$ (o $n_s \leq 2/3\,n$)

Complessità di Heapify:caso peggiore



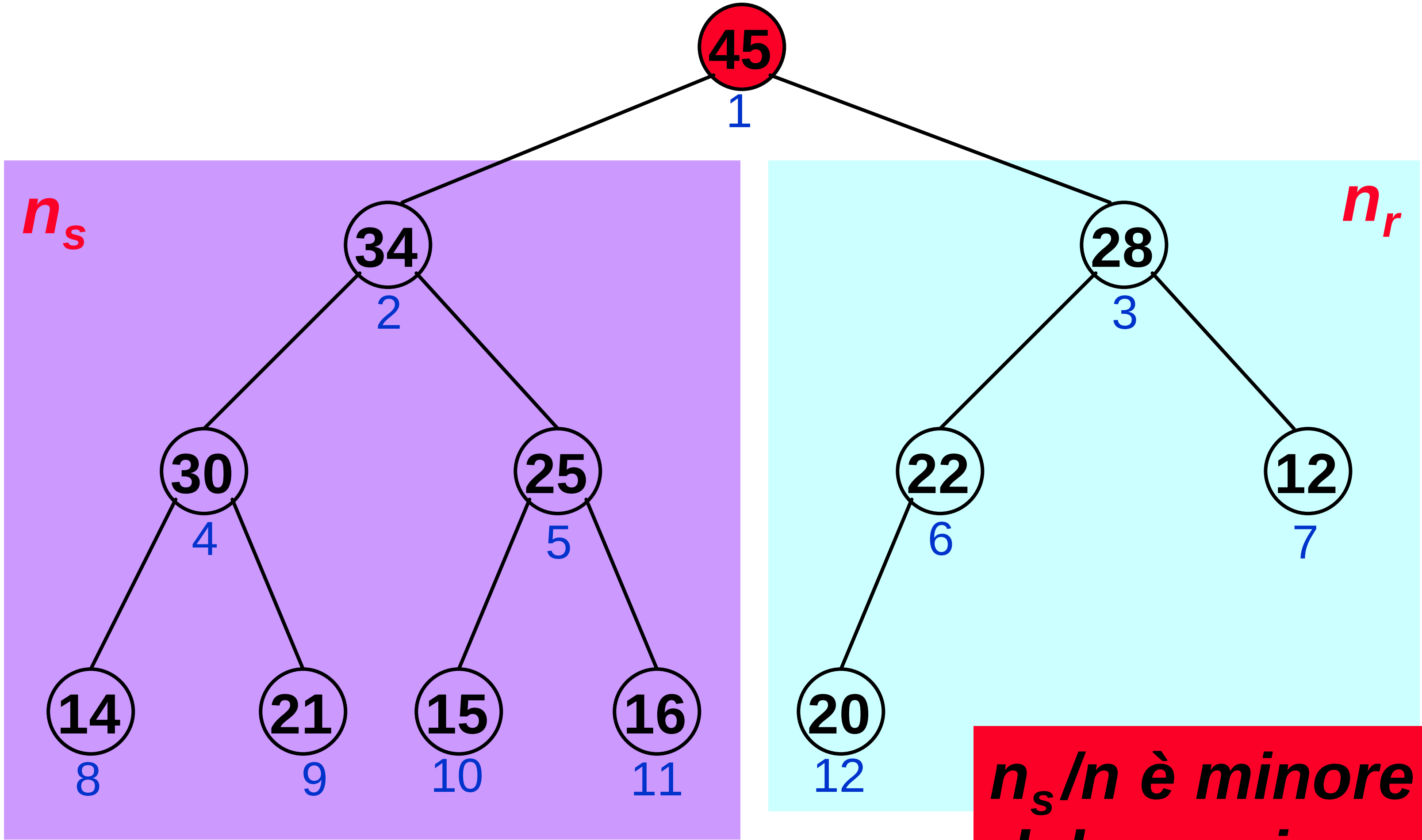
n_s/n ha valore massimo

Complessità di Heapify:caso peggiore



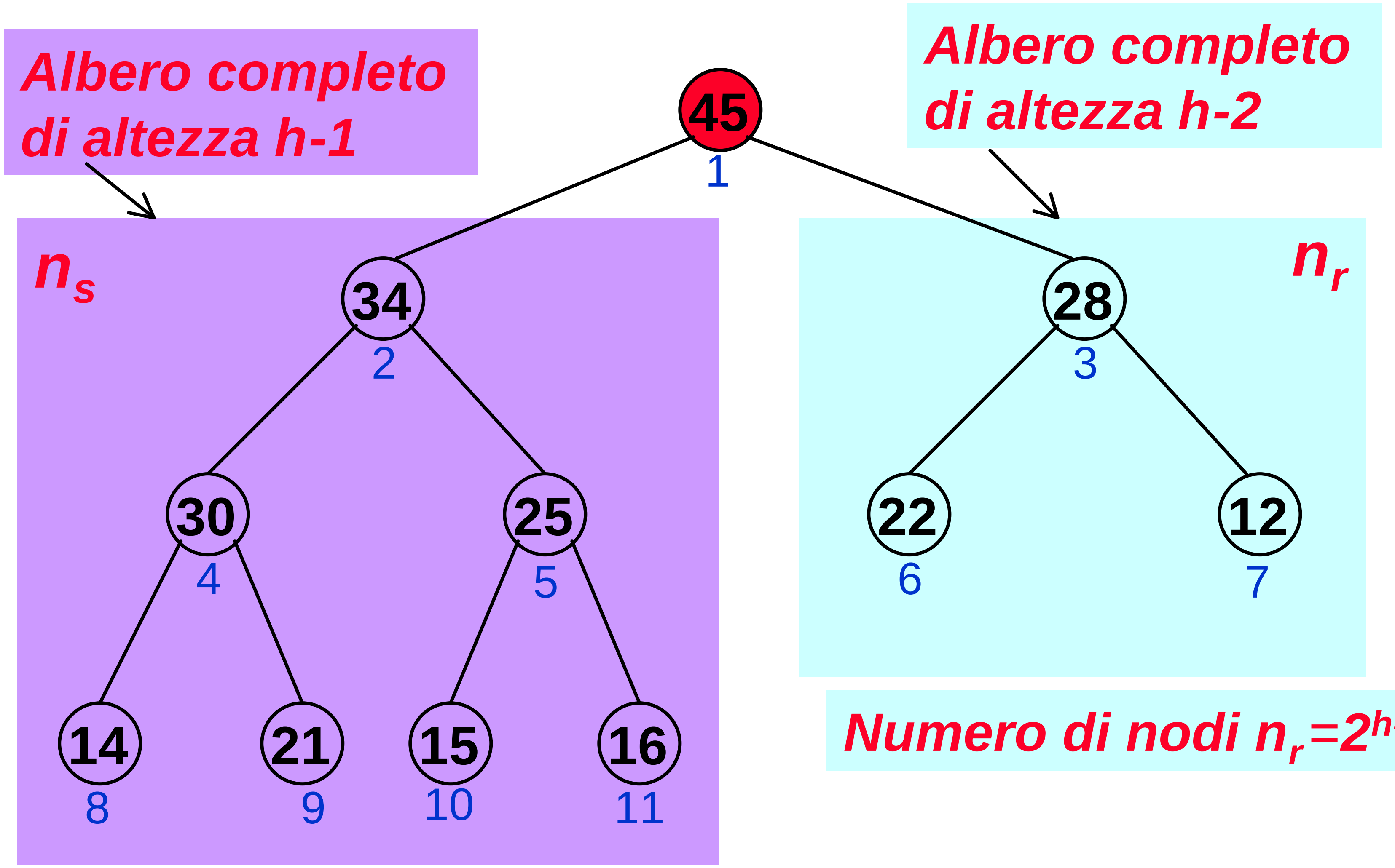
n_s/n è minore del massimo (n_s è più piccolo)

Complessità di Heapify:caso peggiore



n_s/n è minore del massimo (n è più grande)

Complessità di Heapify:caso peggiore

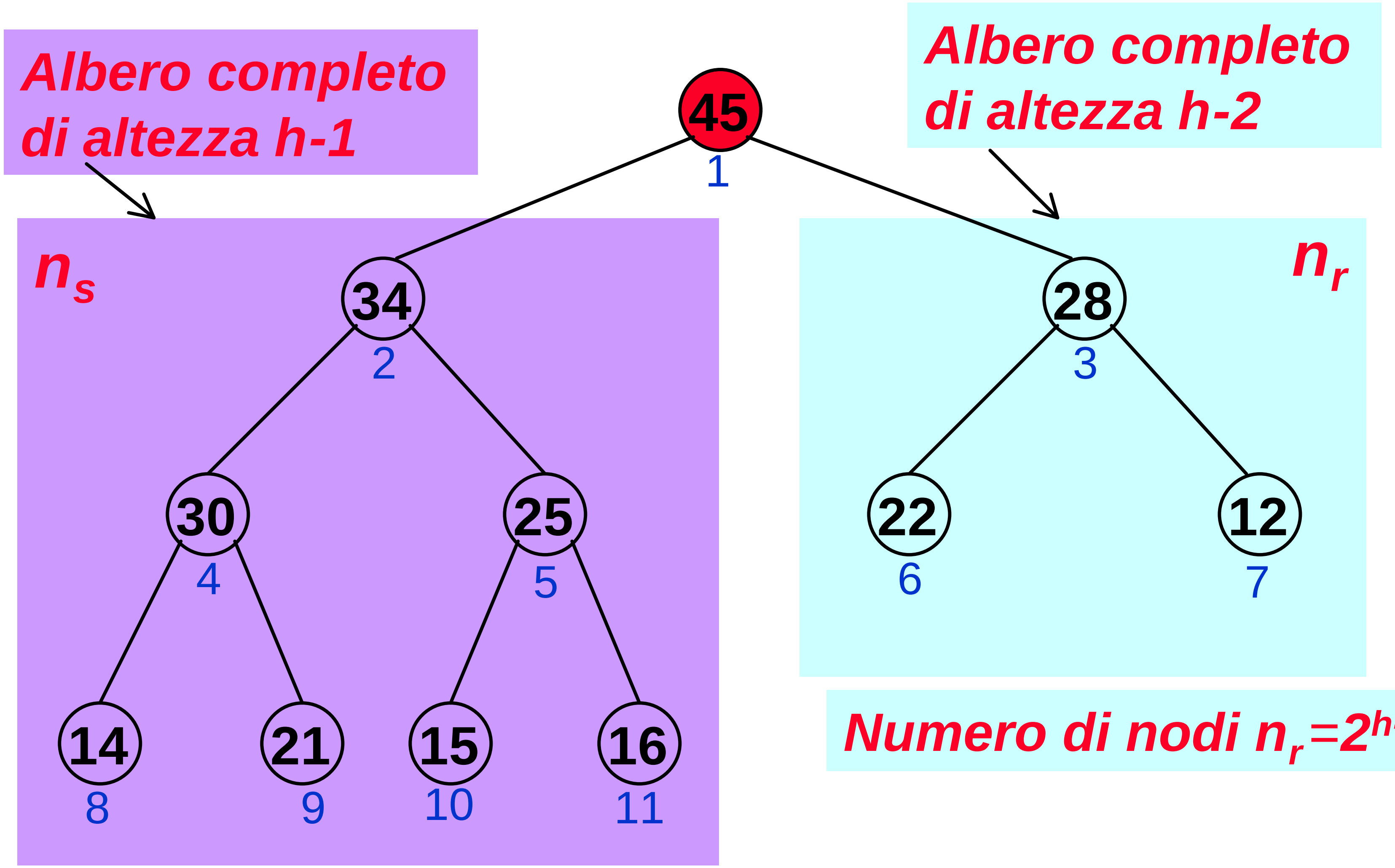


Numero di nodi $n_r = 2^{h-1}-1$

Numero di nodi $n_s = 2^{h}-1$

$$n = 1 + 2^{h}-1 + 2^{h-1}-1 = 3\,2^{h-1} - 1$$

Complessità di Heapify: caso peggiore



Numero di nodi $n_r = 2^{h-1} - 1$

Numero di nodi $n_s = 2^h - 1$

$n_s/n = 2^h - 1/(3\,2^{h-1} - 1)$ £ $2/3$

Complessità di Heapify:caso peggiore

$$T(n) = \max(O(1), \max(O(1), T(?) + O(1))$$

$$\le \max(O(1), \max(O(1), T(2n/3) + O(1))$$

$$\le T(2n/3) + \Theta(1)$$

$$T'(n) = T'(2n/3) + \Theta(1)$$

Proviamo ad applicare il Metodo Iterativo!

$$T'(n) = \Theta(\log n)$$

Complessità di Heapify:caso peggiore

T(n) = *max(O(*1*),max(O(*1*),T(?)* + *O(*1*))*

£ *max(O(*1*),max(O(*1*),T(*2*n/*3*)* + *O(*1*))*

£ *T(*2*n/*3*)* + Q*(*1*)*

Quindi

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

`Heapify` ***impiega tempo proporzionale all’altezza dell’albero su cui opera !***

Complessità di Heapify:caso migliore

$T(n) = T(?) + O(1)$

Nel *caso migliore* *Heapify* ad ogni chiamata ricorsiva, viene eseguito su un numero di nodi che è maggiore di 1/3 del numero di nodi correnti n.

Cioè il numero di nodi n_s del sottoalbero su cui *Heapify* è chiamato ricorsivamente è al più 1/3 n (o $n_s \geq 1/3\, n$)

Complessità di Heapify: caso migliore

$$T(n) = T(?) + O(1)$$

$$\geq T(n/3) + Q(1)$$

$$T'(n) = T'(n/3) + Q(1)$$

Applicando il Metodo Iterativo!

$$T'(n) = Q(\log n)$$ **quindi** $$T(n) = W(\log n)$$

Costruisci Heap: intuizioni

`Costruisci-Heap(`*`A`*`)`: utilizza l'algoritmo `Heapify`, per inserire ogni elemento dell'array in uno *Heap*, risistemando sul posto gli elementi:

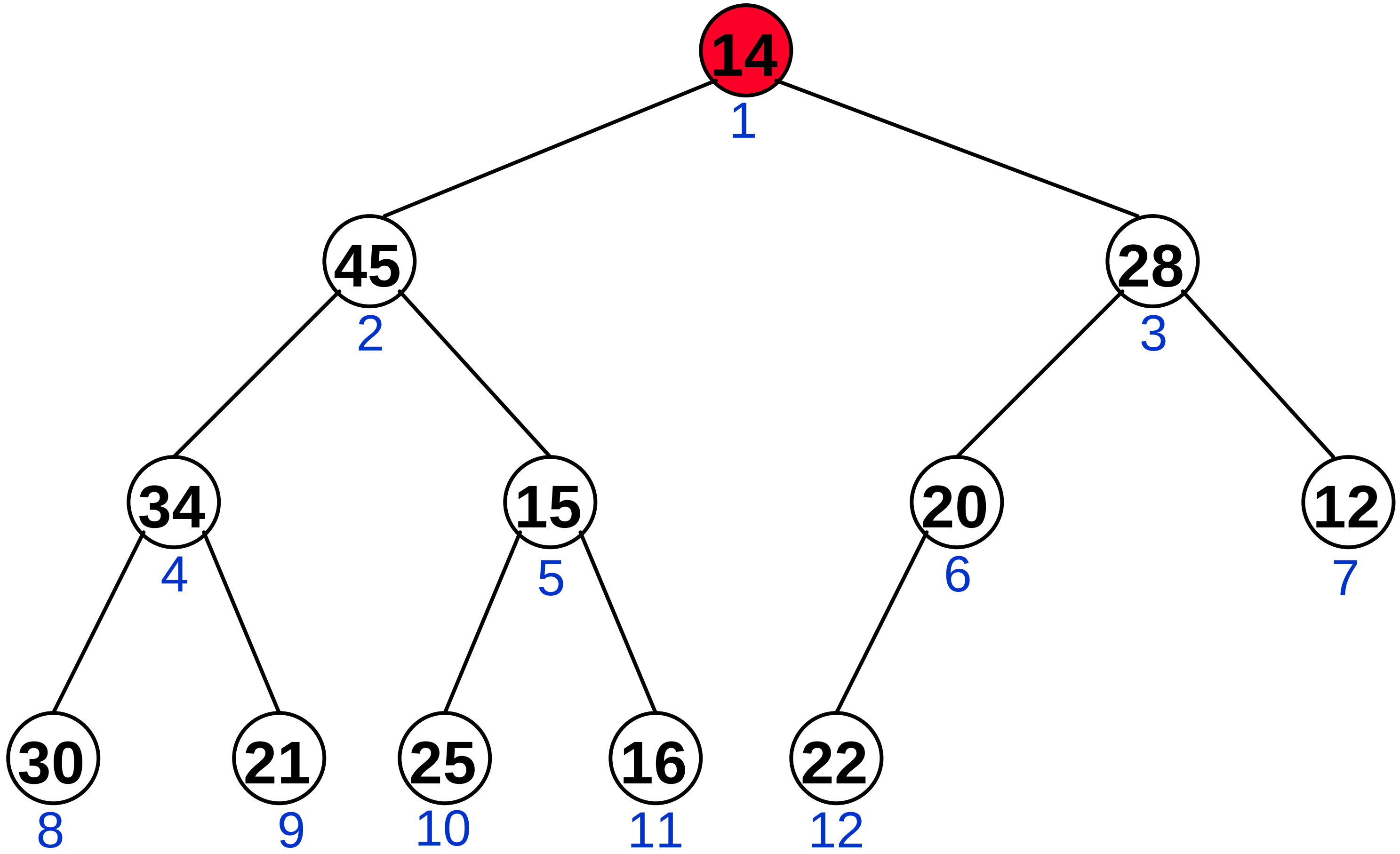
- **gli ultimi é*n*/2ù elementi dell'array sono foglie, cioè radici di sottoalberi vuoti, quindi sono già degli *Heap***
- **è sufficiente inserire nello *Heap* solo i primi ë*n*/2û elementi, utilizzando `Heapify` per ripristinare la proprietà *Heap* sul sottoalbero del nuovo elemento.**

Costruisci Heap

```
Costruisci-Heap(A)
    heapsize[A] = length[A]
    FOR i = ëlength[A]/2û DOWNTO 1
        DO Heapify(A,i)
```

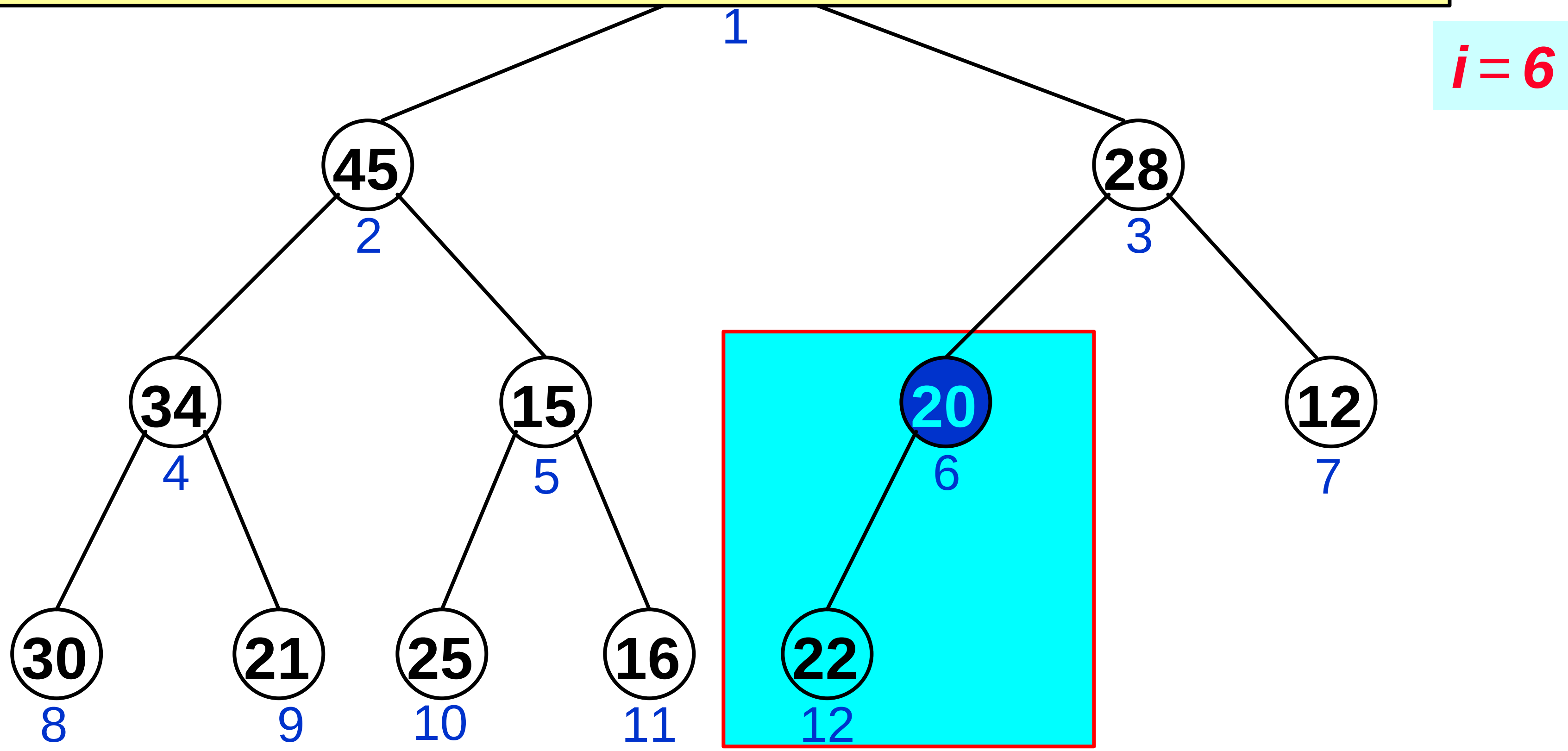
Costruisci Heap

1	2	3	4	5	6	7	8	9	10	11	12
14	**45**	**28**	**34**	**15**	**20**	**12**	**30**	**21**	**25**	**16**	**22**



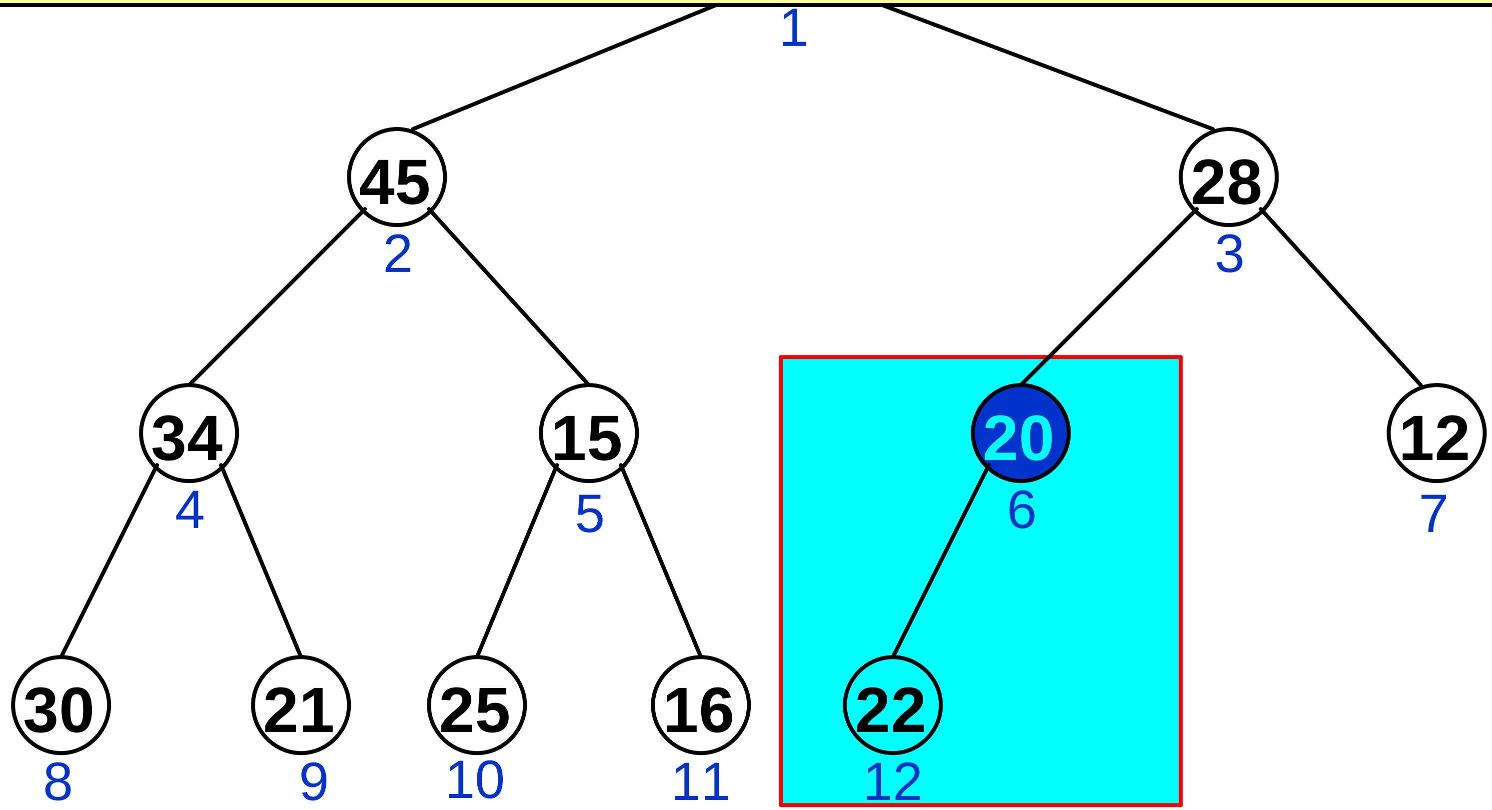
Costruisci Heap

```
Costruisci-Heap(A)
    heapsize[A] = length[A]
    FOR i = ëlength[A]/2û DOWNTO 1
        DO Heapify(A,i)
```



Costruisci Heap

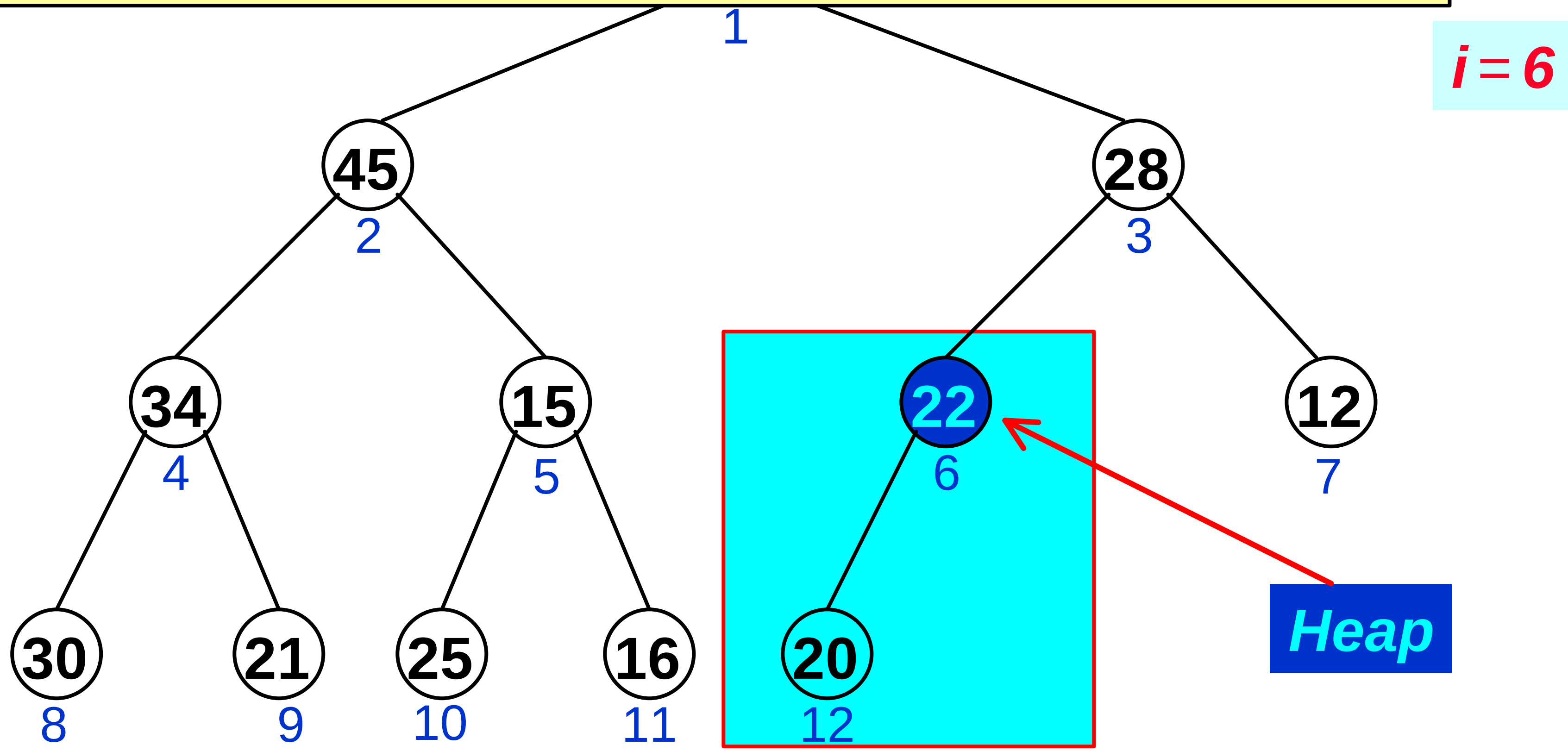
```
Costruisci-Heap(A)
    heapsize[A] = length[A]
    FOR i = ⌊length[A]/2⌋ DOWNTO 1
        DO Heapify(A,i)
```



i = 6

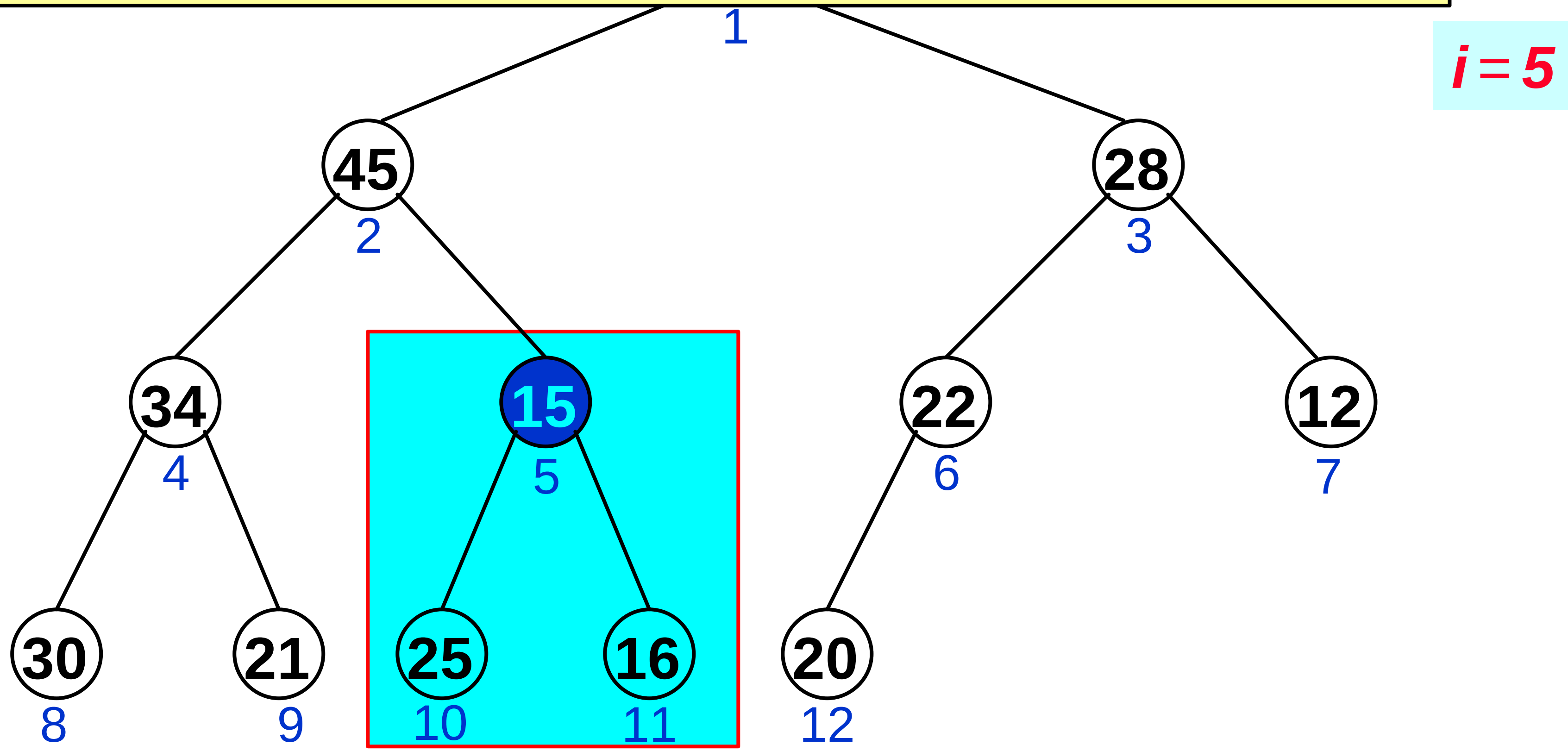
Costruisci Heap

```
Costruisci-Heap(A)
   heapsize[A] = length[A]
   FOR i = ëlength[A]/2û DOWNTO 1
      DO Heapify(A,i)
```



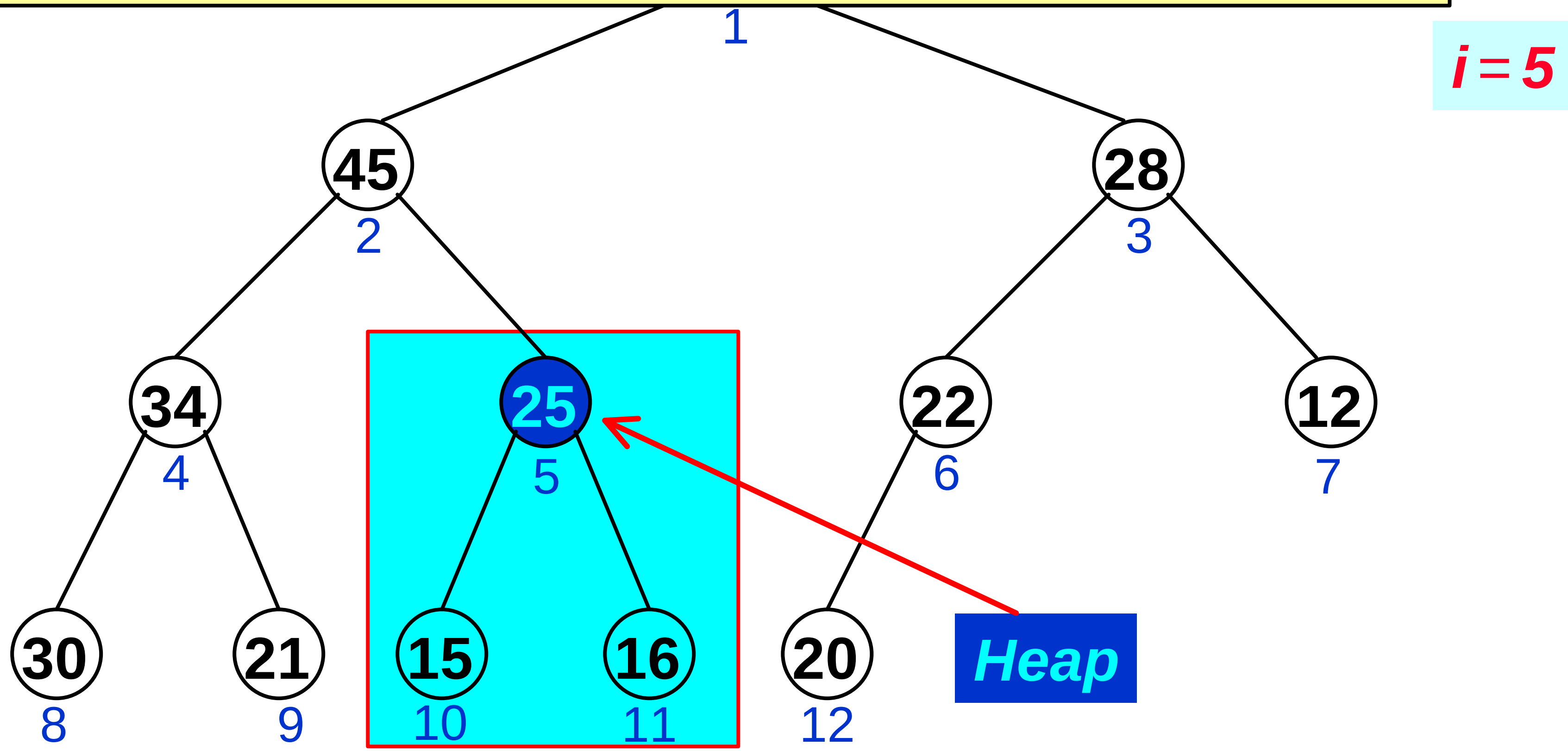
Costruisci Heap

```
Costruisci-Heap(A)
   heapsize[A] = length[A]
   FOR i = ëlength[A]/2û DOWNTO 1
      DO Heapify(A,i)
```



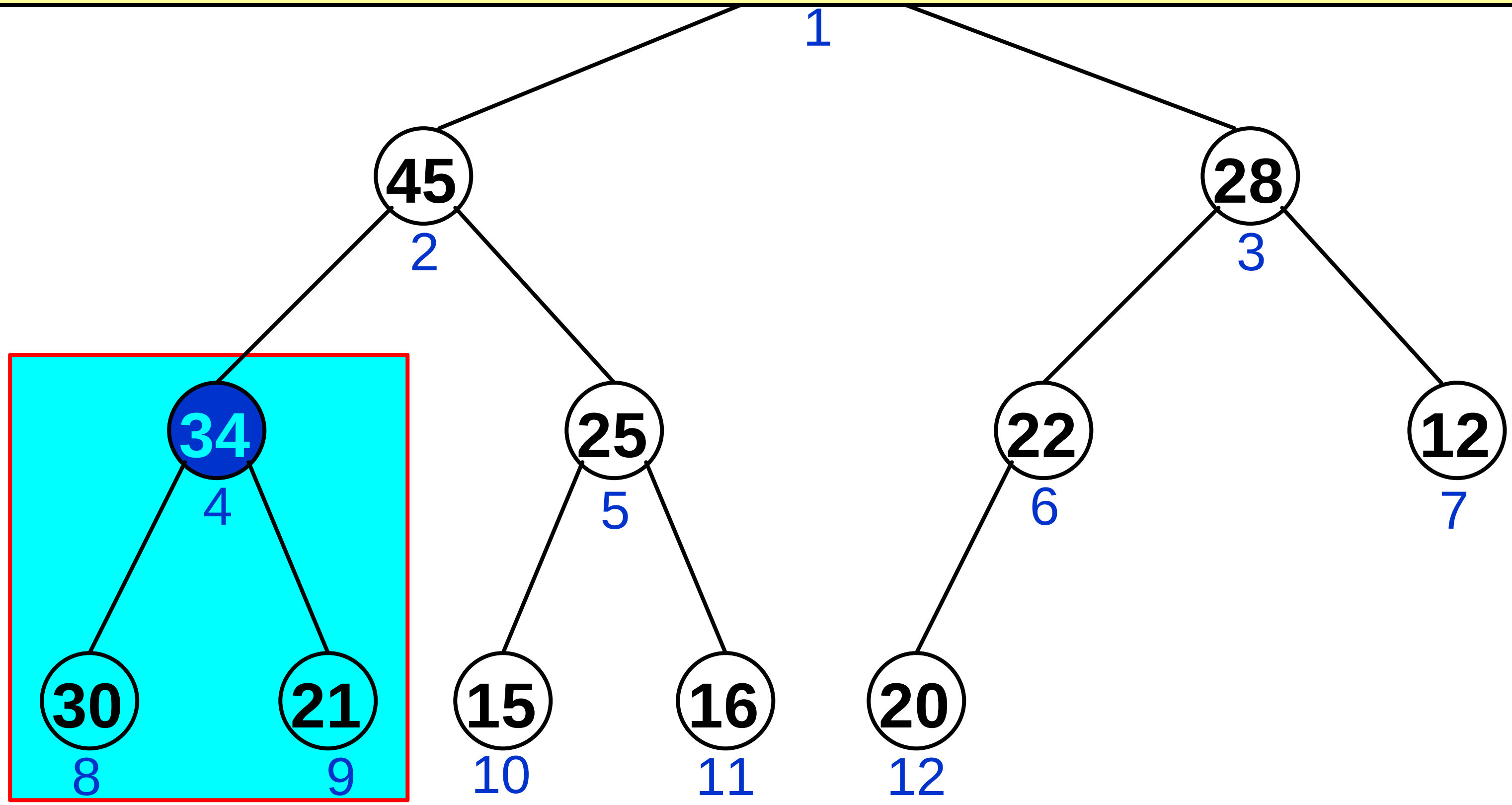
Costruisci Heap

```
Costruisci-Heap(A)
   heapsize[A] = length[A]
   FOR i = ⌊length[A]/2⌋ DOWNTO 1
      DO Heapify(A,i)
```



Costruisci Heap

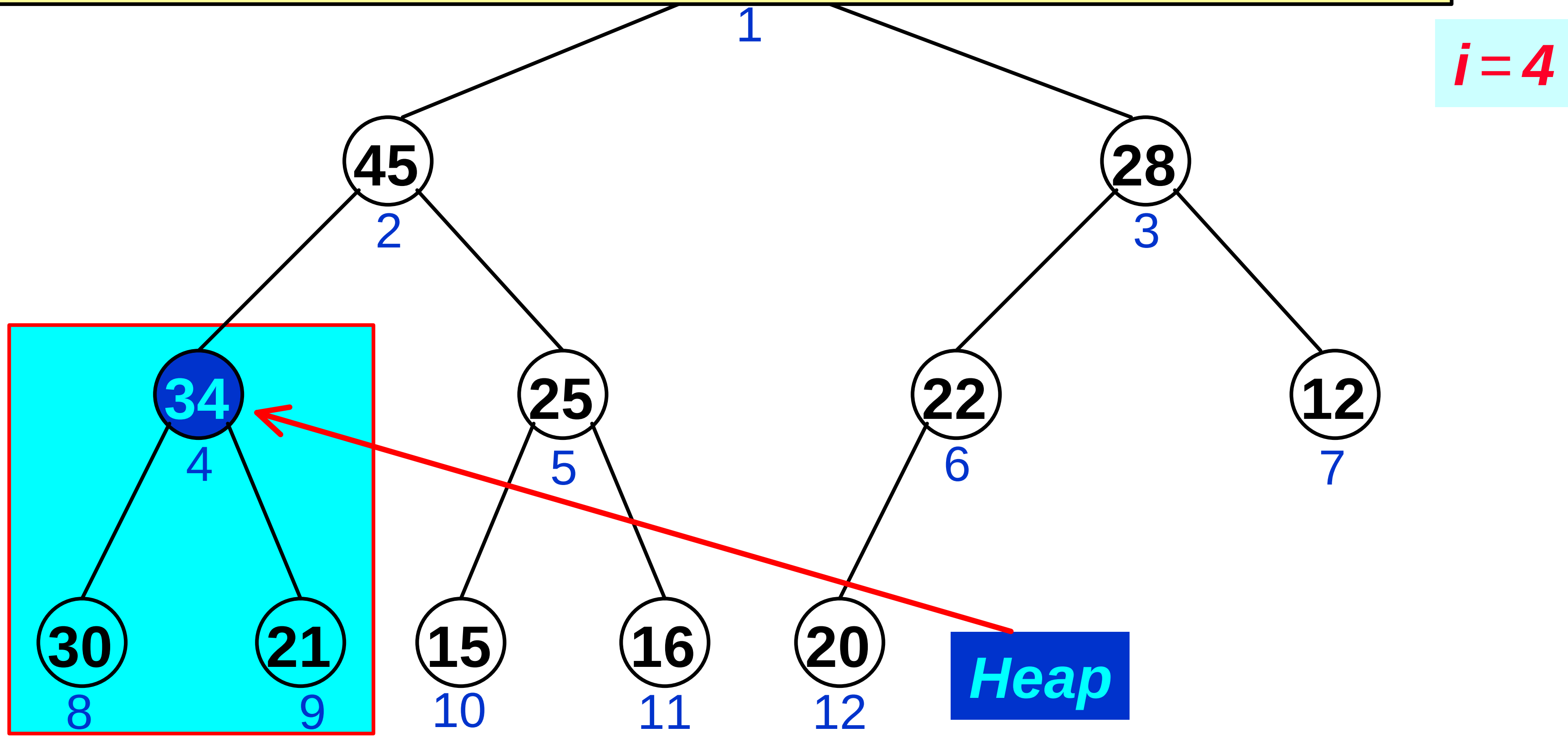
```
Costruisci-Heap(A)
    heapsize[A] = length[A]
    FOR i = ëlength[A]/2û DOWNTO 1
        DO Heapify(A,i)
```



i = 4

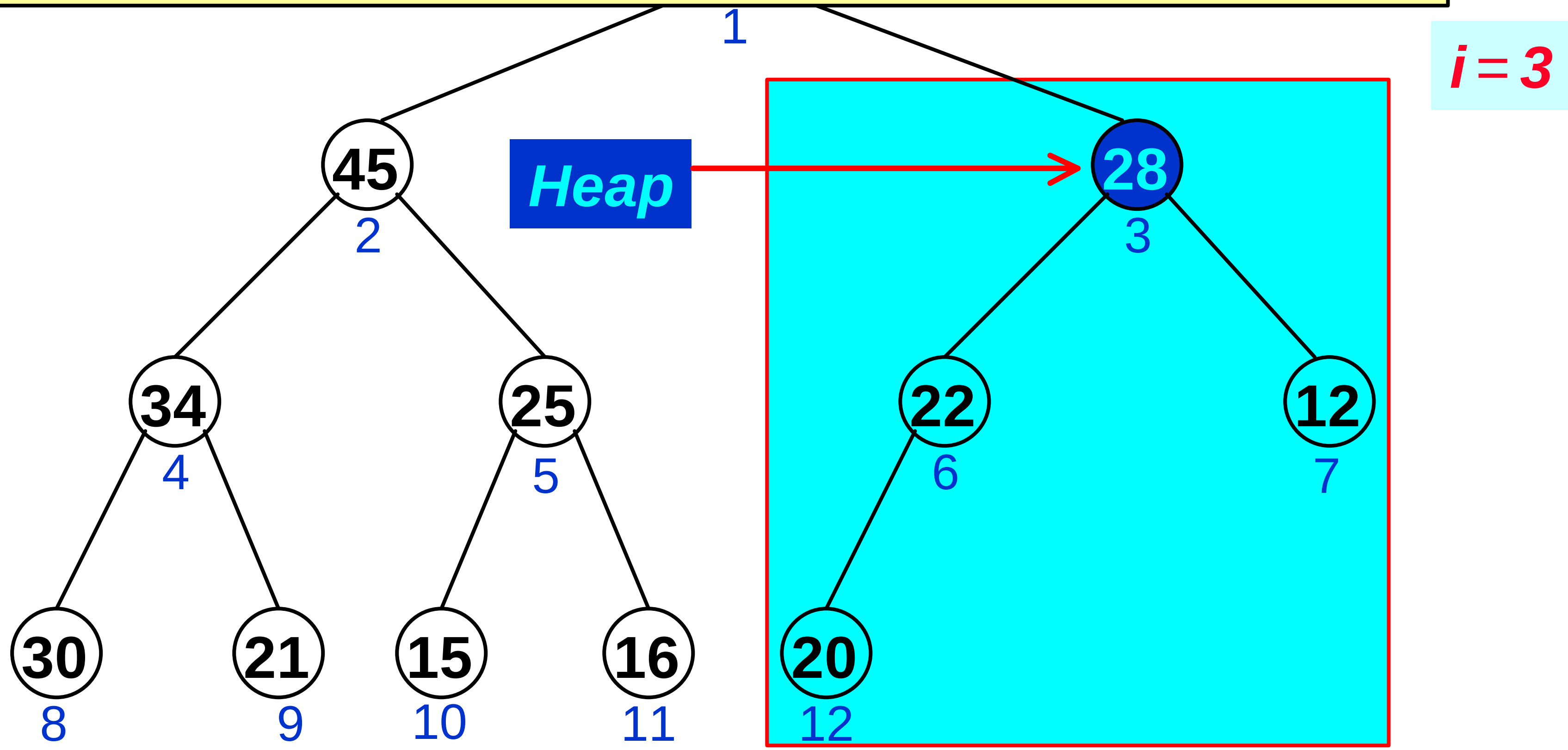
Costruisci Heap

```
Costruisci-Heap(A)
    heapsize[A] = length[A]
    FOR i = ëlength[A]/2û DOWNTO 1
        DO Heapify(A,i)
```



Costruisci Heap

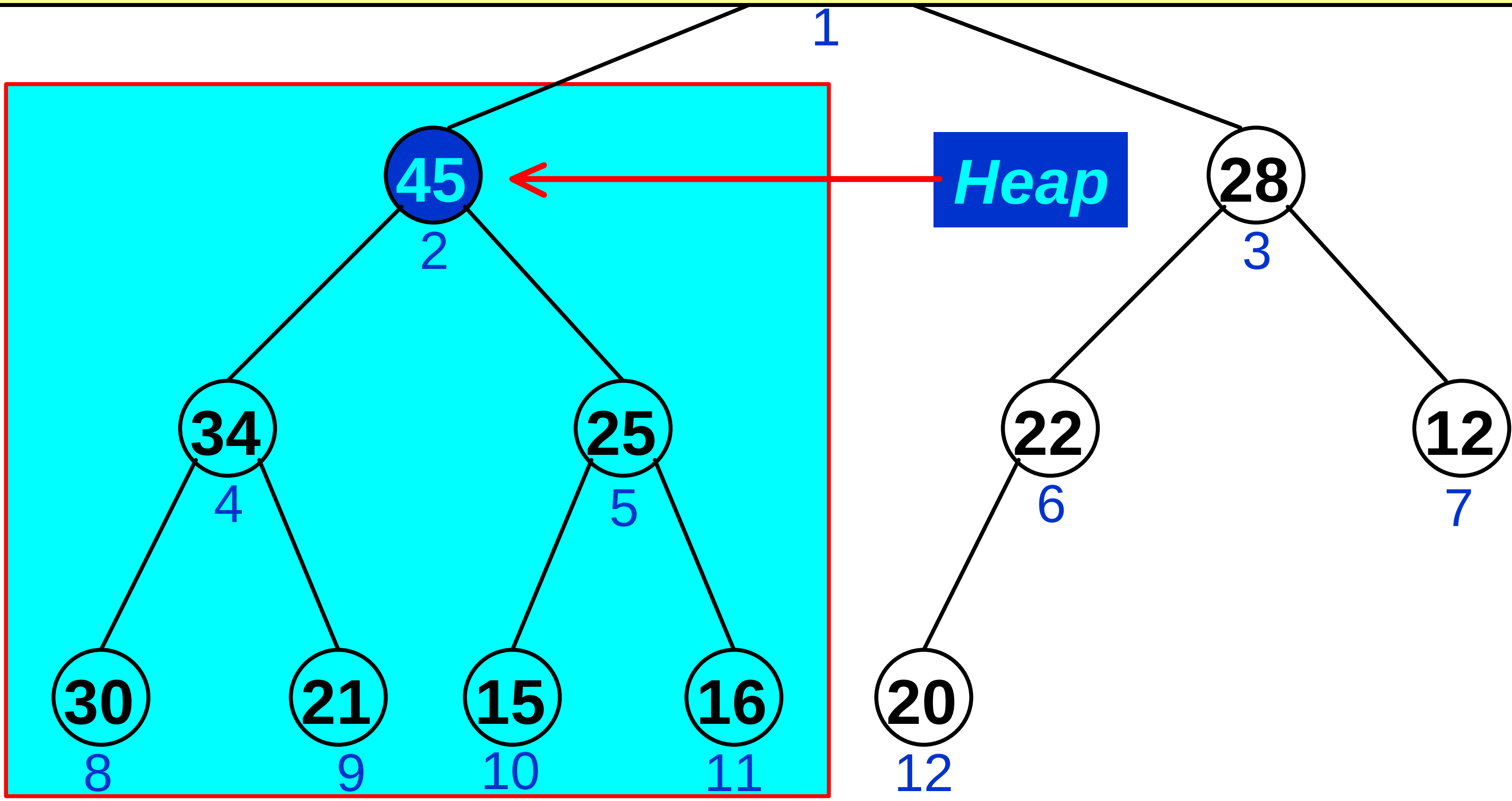
```
Costruisci-Heap(A)
   heapsize[A] = length[A]
   FOR i = ëlength[A]/2û DOWNTO 1
      DO Heapify(A,i)
```



Costruisci Heap

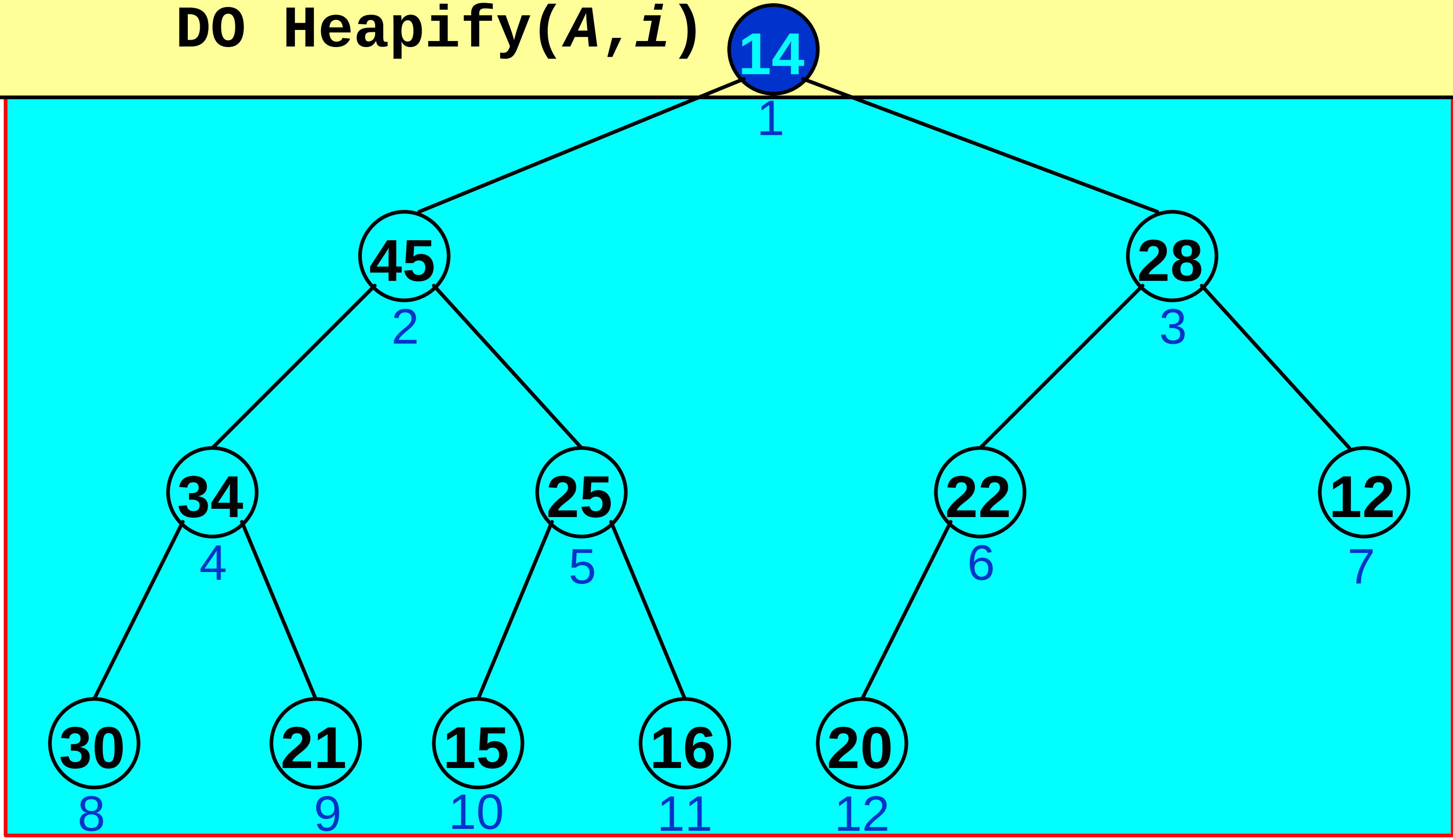
```
Costruisci-Heap(A)
   heapsize[A] = length[A]
   FOR i = ëlength[A]/2û DOWNTO 1
      DO Heapify(A,i)
```

i = 2



Costruisci Heap

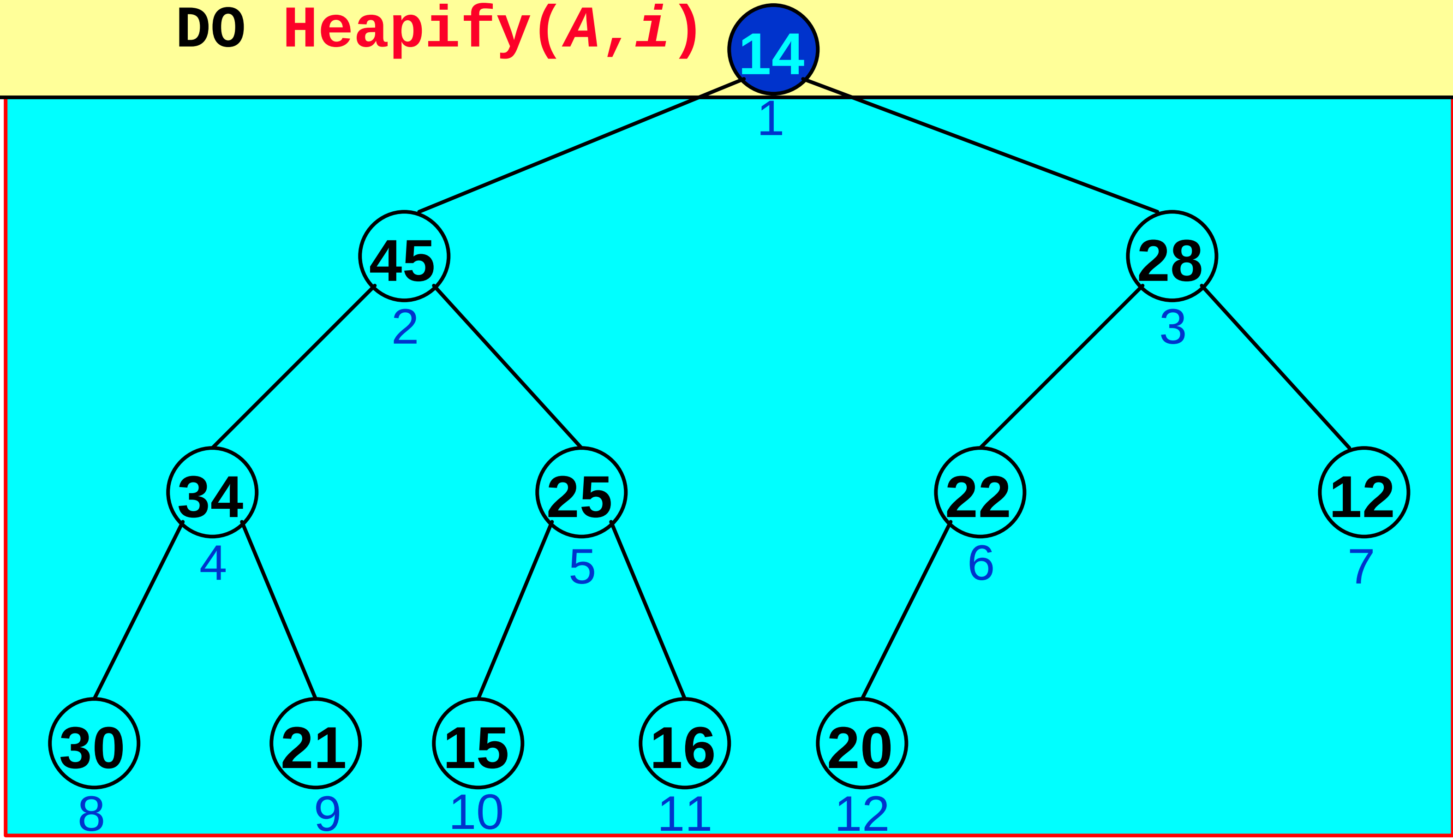
```
Costruisci-Heap(A)
    heapsize[A] = length[A]
    FOR i = ⌊length[A]/2⌋ DOWNTO 1
        DO Heapify(A,i)
```



i = 1

Costruisci Heap

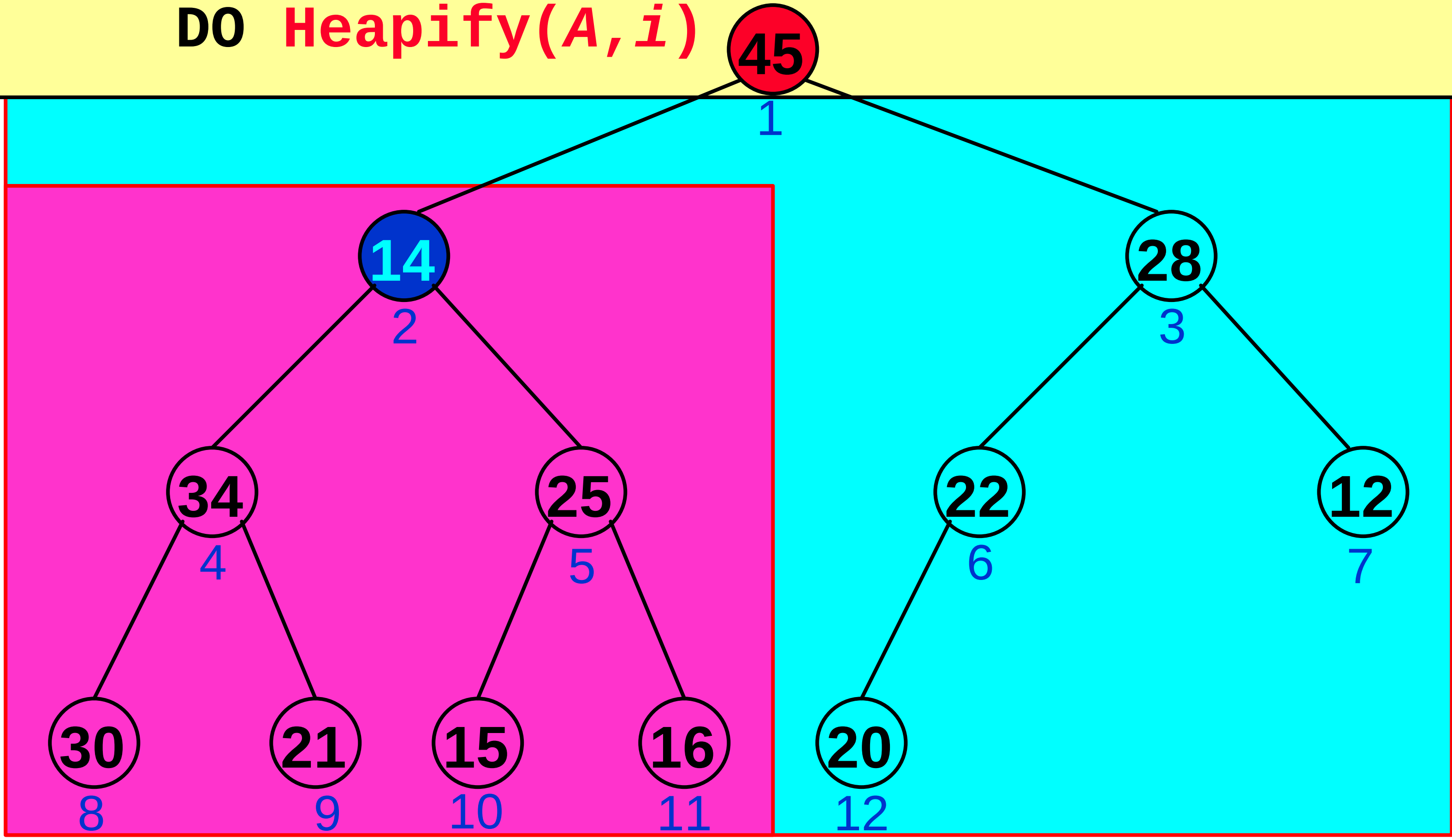
```
Costruisci-Heap(A)
   heapsize[A] = length[A]
   FOR i = ëlength[A]/2û DOWNTO 1
      DO Heapify(A,i)
```



i = 1

Costruisci Heap

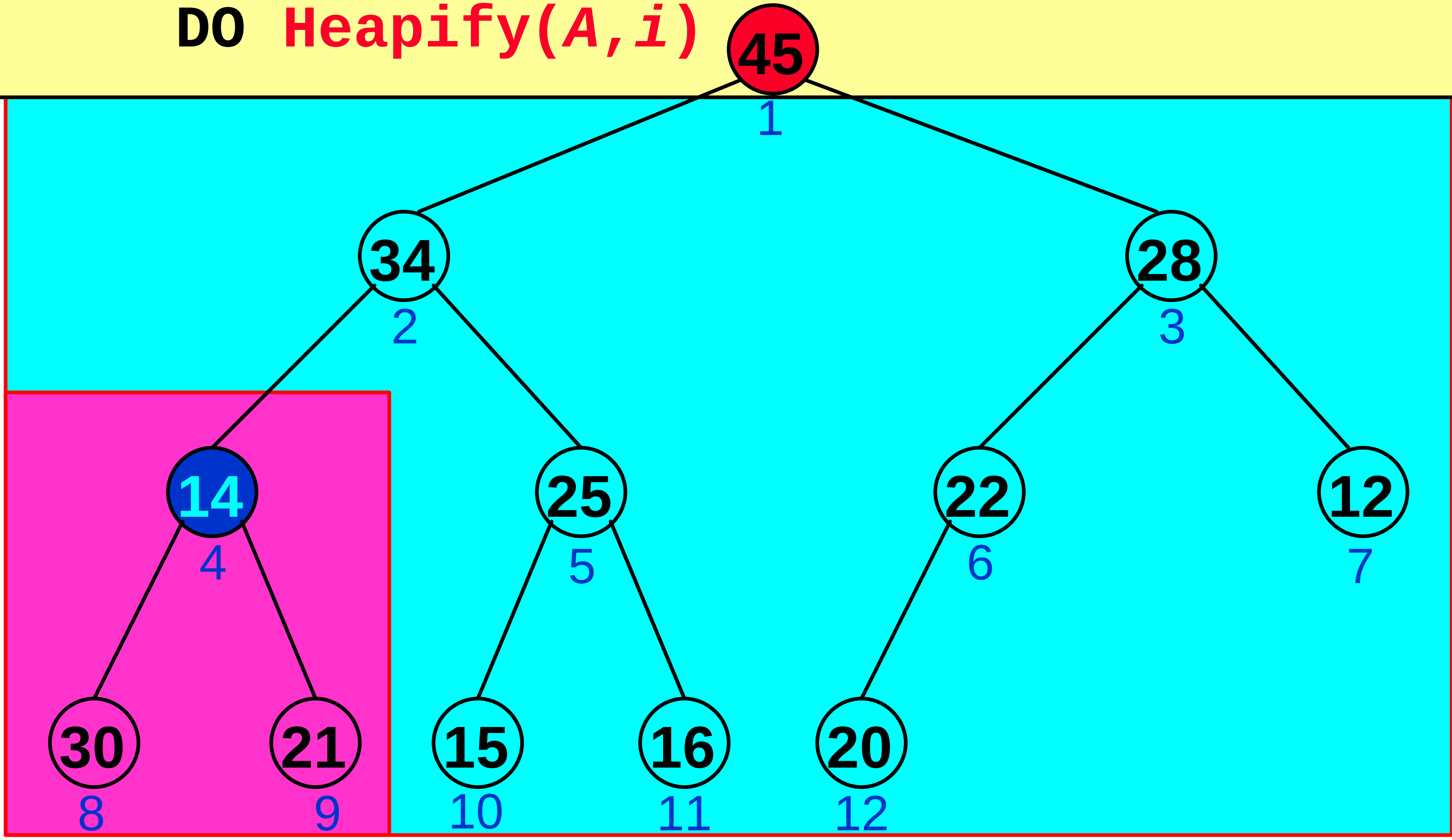
```
Costruisci-Heap(A)
    heapsize[A] = length[A]
    FOR i = ëlength[A]/2û DOWNTO 1
        DO Heapify(A,i)
```



i = 1

Costruisci Heap

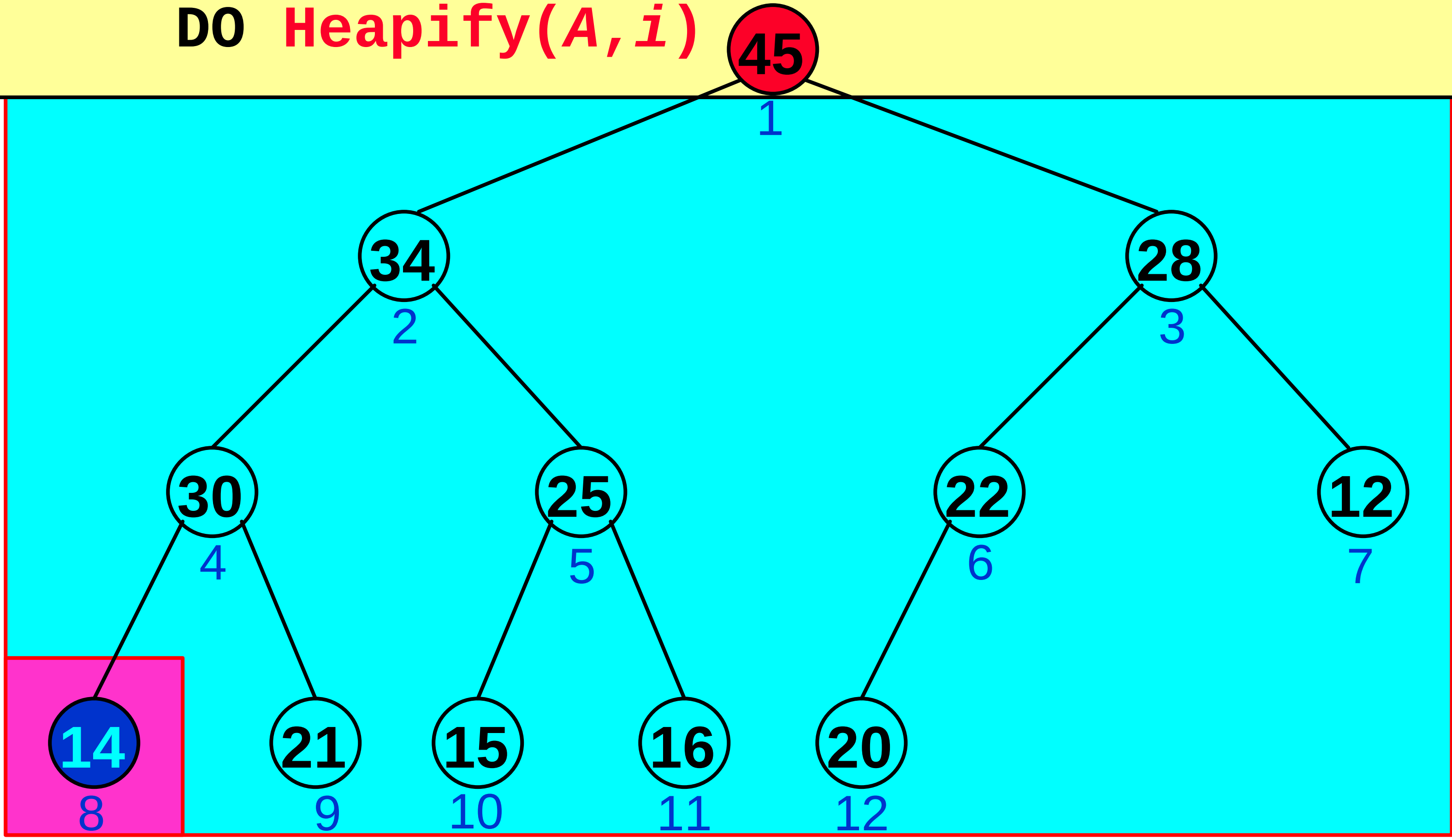
```
Costruisci-Heap(A)
    heapsize[A] = length[A]
    FOR i = ⌊length[A]/2⌋ DOWNTO 1
        DO Heapify(A,i)
```



i = 1

Costruisci Heap

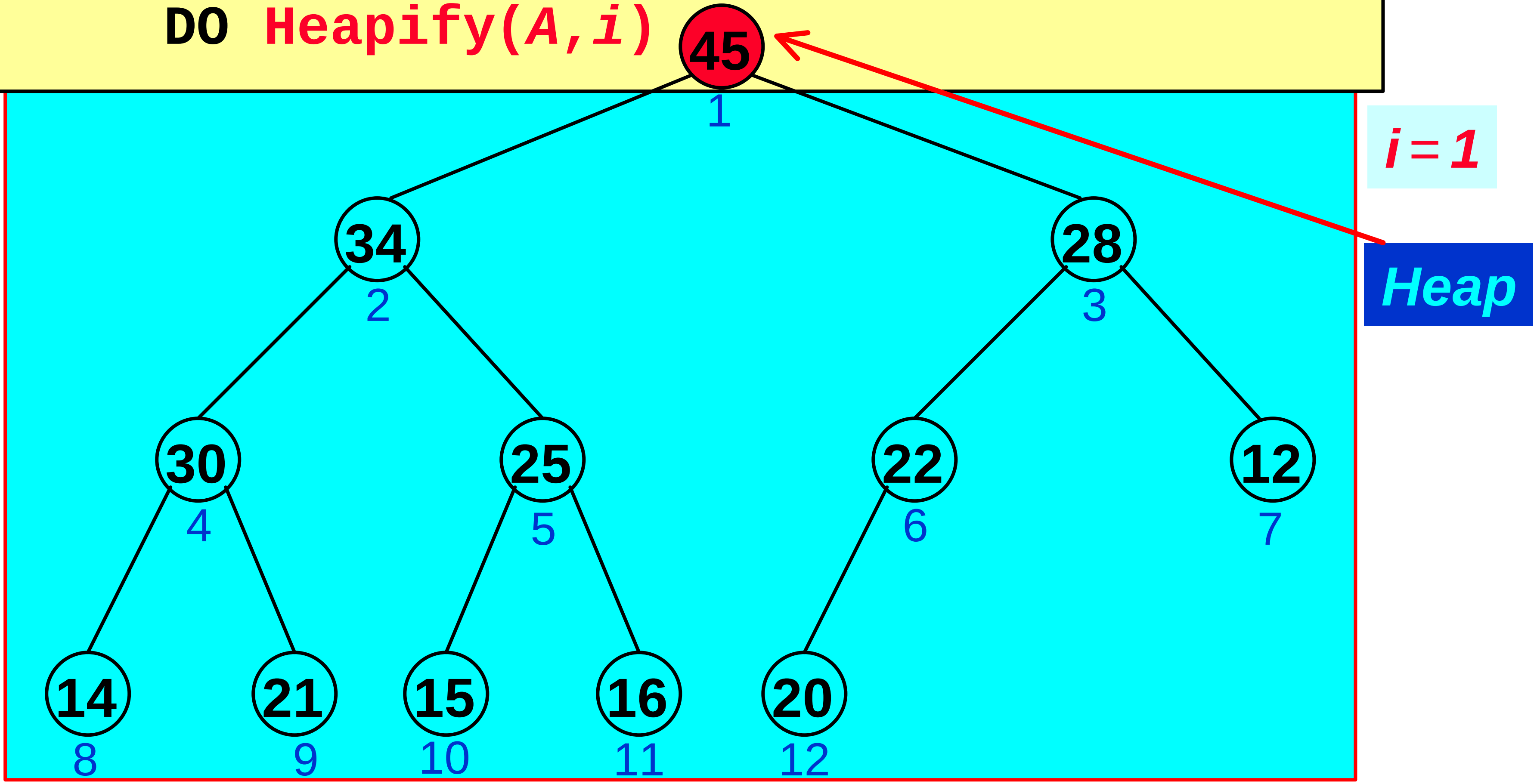
```
Costruisci-Heap(A)
    heapsize[A] = length[A]
    FOR i = ëlength[A]/2û DOWNTO 1
        DO Heapify(A,i)
```



i = 1

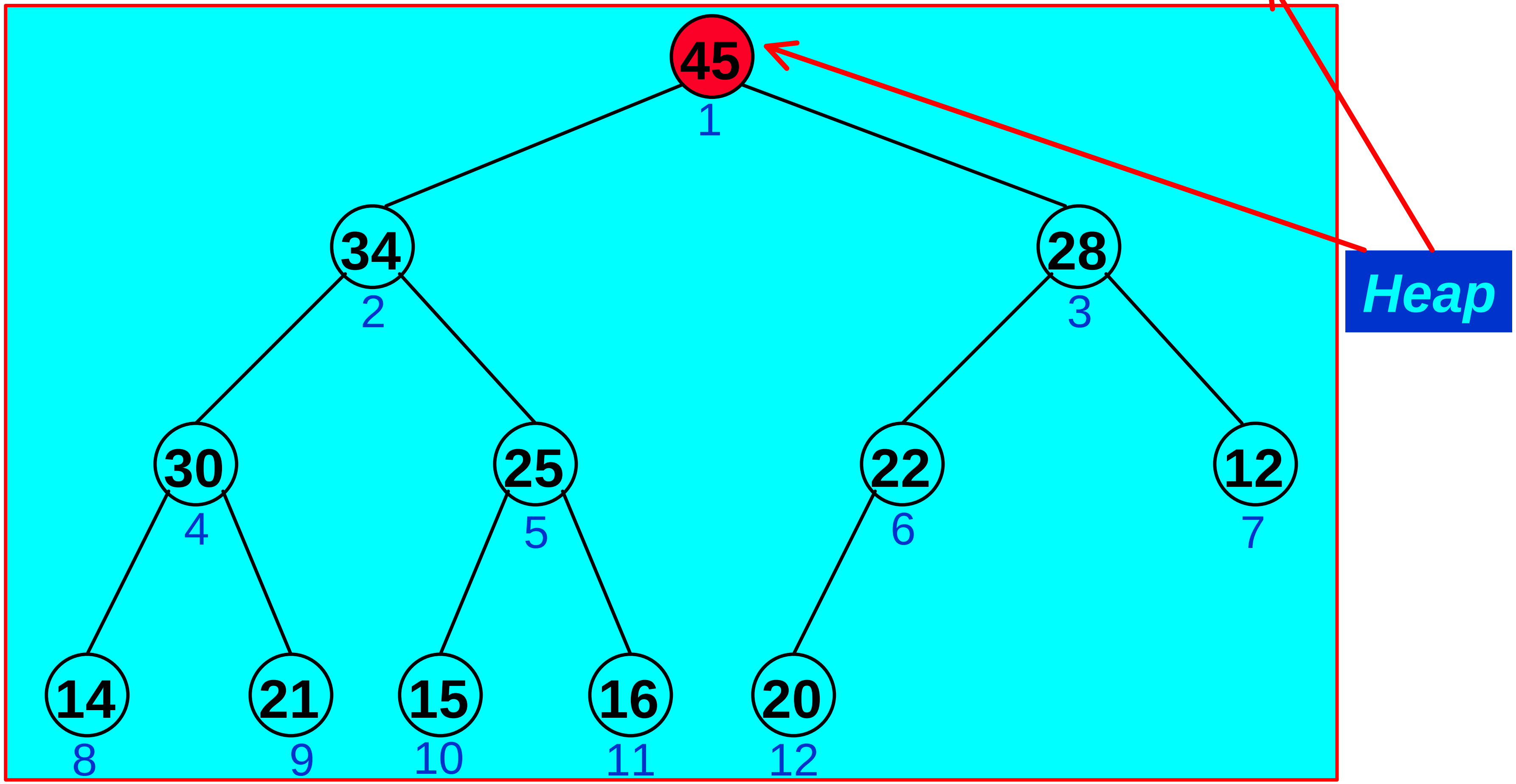
Costruisci Heap

```
Costruisci-Heap(A)
   heapsize[A] = length[A]
   FOR i = ëlength[A]/2û DOWNTO 1
      DO Heapify(A,i)
```



Costruisci Heap

1	2	3	4	5	6	7	8	9	10	11	12
45	**34**	**28**	**30**	**25**	**22**	**12**	**14**	**21**	**15**	**16**	**20**



Complessità di Costruisci Heap

```
Costruisci-Heap(A)
   heapsize[A] = length[A]               } = O(1)
   FOR i = ëlength[A]/2û DOWNTO 1        }
      DO Heapify(A,i)                    } = O(?)
```

Complessità di Costruisci Heap

$T(n) = \max(O(1), O(?)) = \max(O(1), O(f(n)))$

Poiché `Heapify` viene chiamata $n/2$ volte si potrebbe ipotizzare

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

e quindi

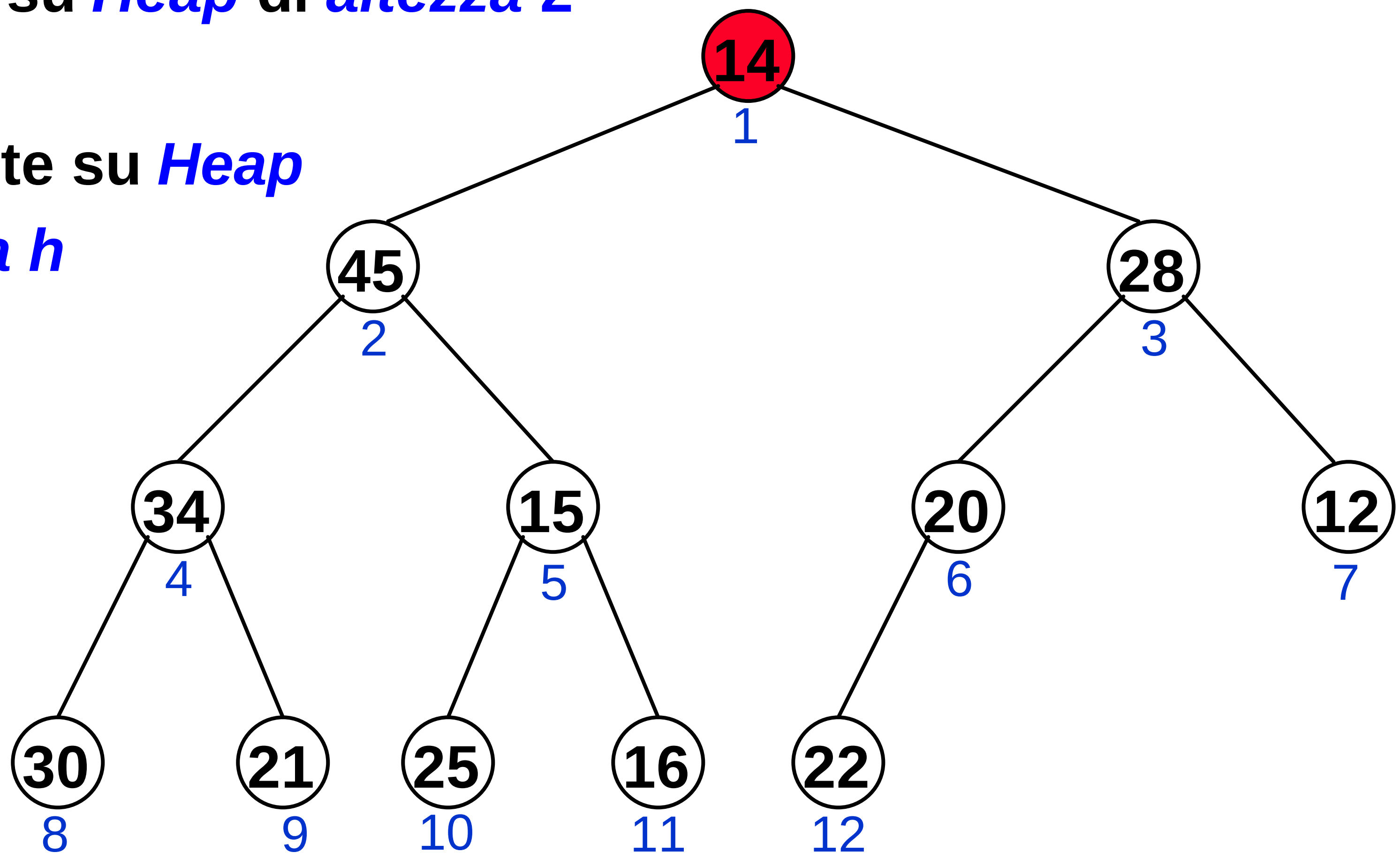
$$T(n) = \max(O(1), O(n \log n)) = O(n \log n)$$

ma....

Complessità di Costruisci Heap

Costruisci-Heap **chiama** ***Heapify***

- ***n*/2 volte su *Heap* di *altezza* 0 (non eseguito)**
- ***n*/4 volte su *Heap* di *altezza* 1**
- ***n*/8 volte su *Heap* di *altezza* 2**
- **...**
- **$n/2^{h+1}$ volte su *Heap* di *altezza h***



Complessità di Costruisci Heap

T(n) = max(O(1), O(?)) = max(O(1), O(f(n)))

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

Costruisci-Heap **chiama** ***Heapify***

- ***n*/2 volte su *Heap* di *altezza* 0 (in reltà non eseguito)**
- ***n*/4 volte su *Heap* di *altezza* 1**
- ***n*/8 volte su *Heap* di *altezza* 2**
- **…**
- ***n*/2^{h+1} volte su *Heap* di *altezza h***

Complessità di Costruisci Heap

T(n) = max(O(1), O(?)) = max(O(1), O(f(n)))

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

$$= O\left(\frac{n}{2} \sum_{h=0}^{\lfloor \log n \rfloor} h / 2^{h} \right)$$

Complessità di Costruisci Heap

T(n) = max(O(1), O(?)) = max(O(1), O(f(n)))

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

$$= O\left(\frac{n}{2} \sum_{h=0}^{\lfloor \log n \rfloor} h / 2^{h} \right)$$

$$= O\left(\frac{n}{2} \sum_{h=0}^{\infty} h / 2^{h} \right)$$

Complessità di Costruisci Heap

T(n) = max(O(1), O(?)) = max(O(1), O(f(n)))

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

$$= O\left(\frac{n}{2} \sum_{h=0}^{\lfloor \log n \rfloor} h / 2^h \right)$$

$$= O\left(\frac{n}{2} \sum_{h=0}^{\infty} h / 2^h \right)$$

$$= O\left(2n / 2\right)$$

$$\sum_{h=0}^{\infty} h x^h = \frac{x}{(1-x)^2}$$

$$\frac{x}{(1-x)^2} = 2$$

$x = 1/2 \leq 1$

Complessità di Costruisci Heap

T(n) = max(O(1), O(?)) = max(O(1), O(f(n)))

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

$$= O\left(\frac{n}{2} \sum_{h=0}^{\lfloor \log n \rfloor} h/2^h \right)$$

$$= O\left(\frac{n}{2} \sum_{h=0}^{\infty} h/2^h \right)$$

$$= O(n)$$

Complessità di Costruisci Heap

T(n) = max(O(1), O(?)) = max(O(1), O(f(n)))

$$f(n) = O(n)$$

T(n) = max(O(1), O(n)) = O(n)

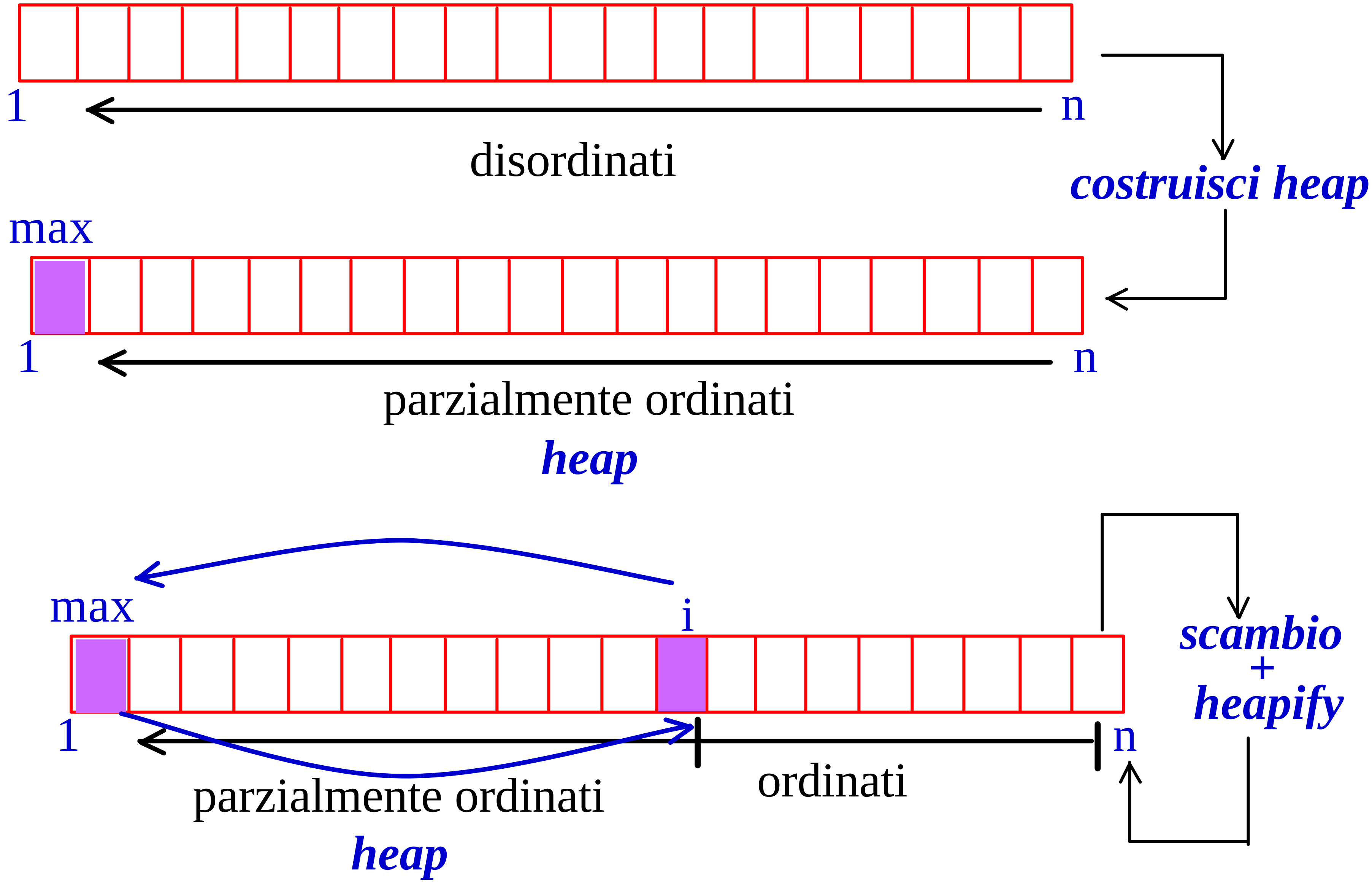
*Costruire uno **Heap** di*
***n** elementi è poco costoso,*
*al più costa **O(n)** !*

Heap Sort: intuizioni

`Heap-Sort`: è una variazione di `Select-sort` in cui la ricerca dell'elemento massimo è facilitata dal mantenimento della sequenza in uno heap:

- **si costruisce uno *Heap* a partire dall'array non ordinato in input.**
- **viene sfruttata la proprietà degli *Heap* per cui la radice `A[1]` dello *Heap* è sempre il massimo:**
  - **scandisce tutti gli elementi dell'array a partire dall'ultimo e ad ogni iterazione**
    - **la radice `A[1]` viene scambiata con l'elemento nell'ultima posizione corrente dello *Heap***
    - **viene ridotta la dimensione dello *Heap* e**
    - **ripristinato lo *Heap* con `Heapify`**

Heap Sort: intuizioni



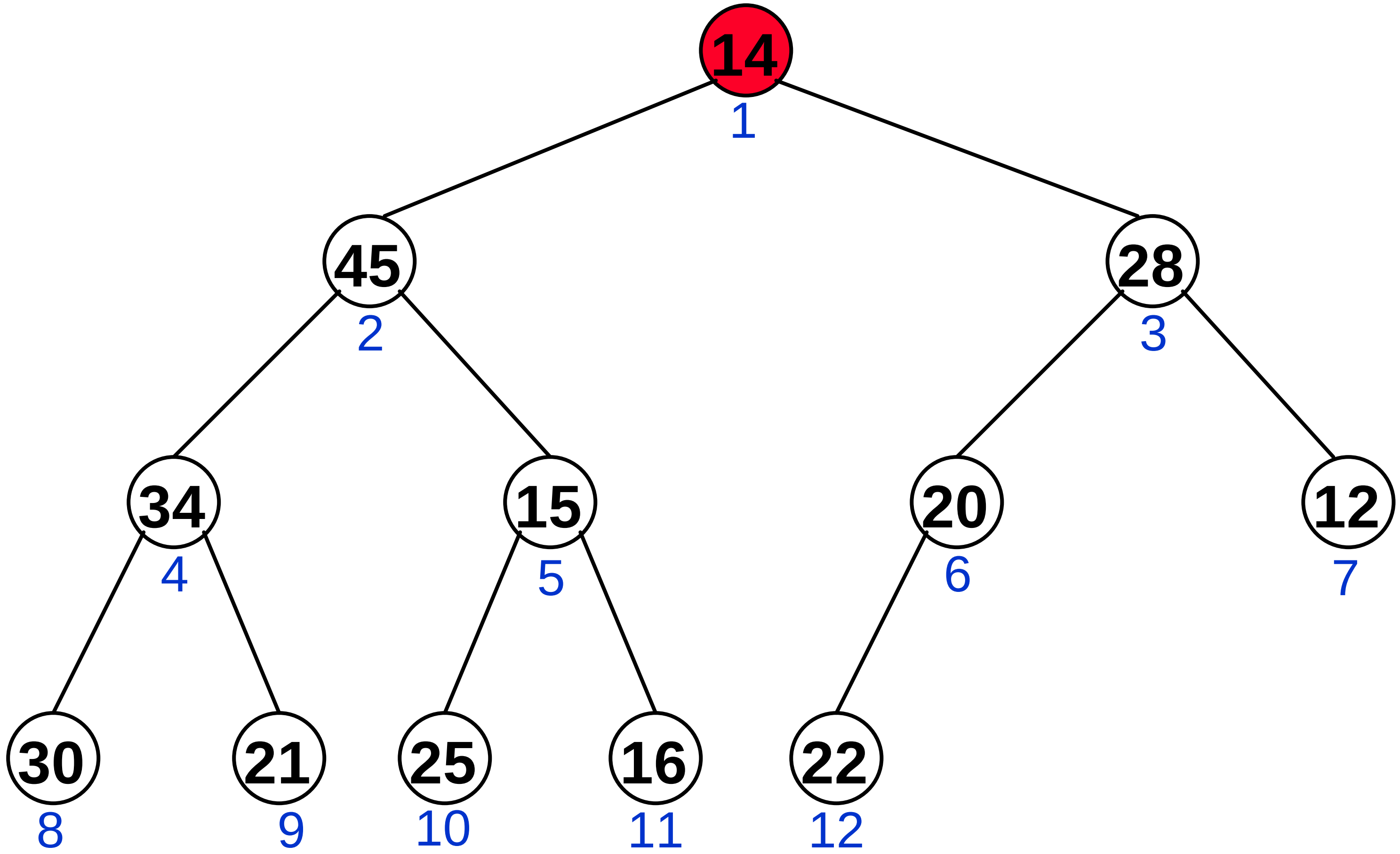
Heap Sort

```
Select-Sort(A)
   FOR i =  length[A] DOWNTO 2
      DO  max = Findmax(A,i)
          "scambia A[max] e A[i]"
```

```
Heap-Sort(A)
   Costruisci-Heap(A)
   FOR i = length[A] DOWNTO 2
      DO /* elemento massimo in A[1] */
          "scambia A[1] e A[i]"
          /* ripristina lo heap */
          heapsize[A] = heapsize[A]-1
          Heapify(A,1)
```

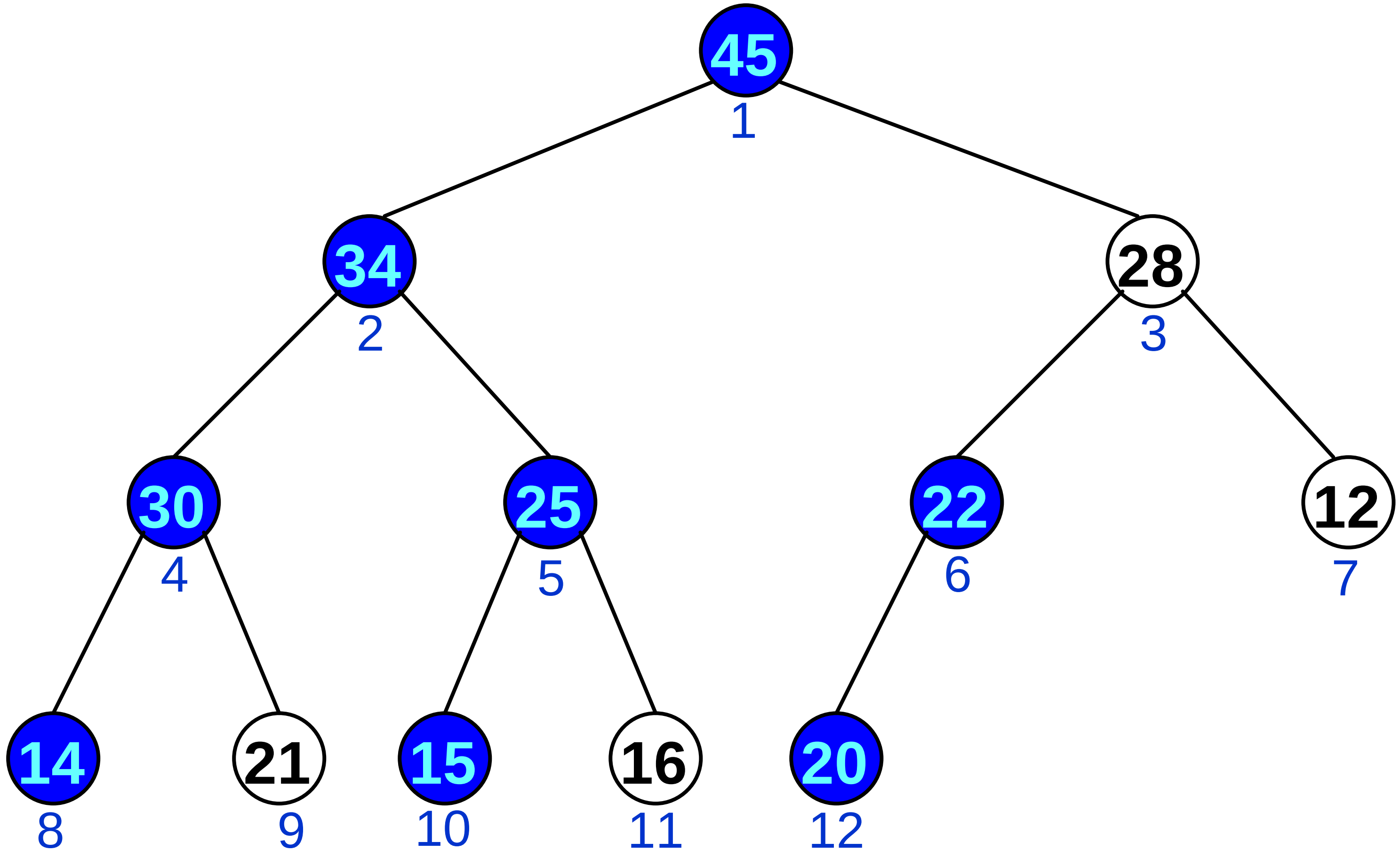
Heap Sort

```
Heap-Sort(A)
    Costruisci-Heap(A)
    …
```



Heap Sort

```
Heap-Sort(A)
    Costruisci-Heap(A)
    …
```

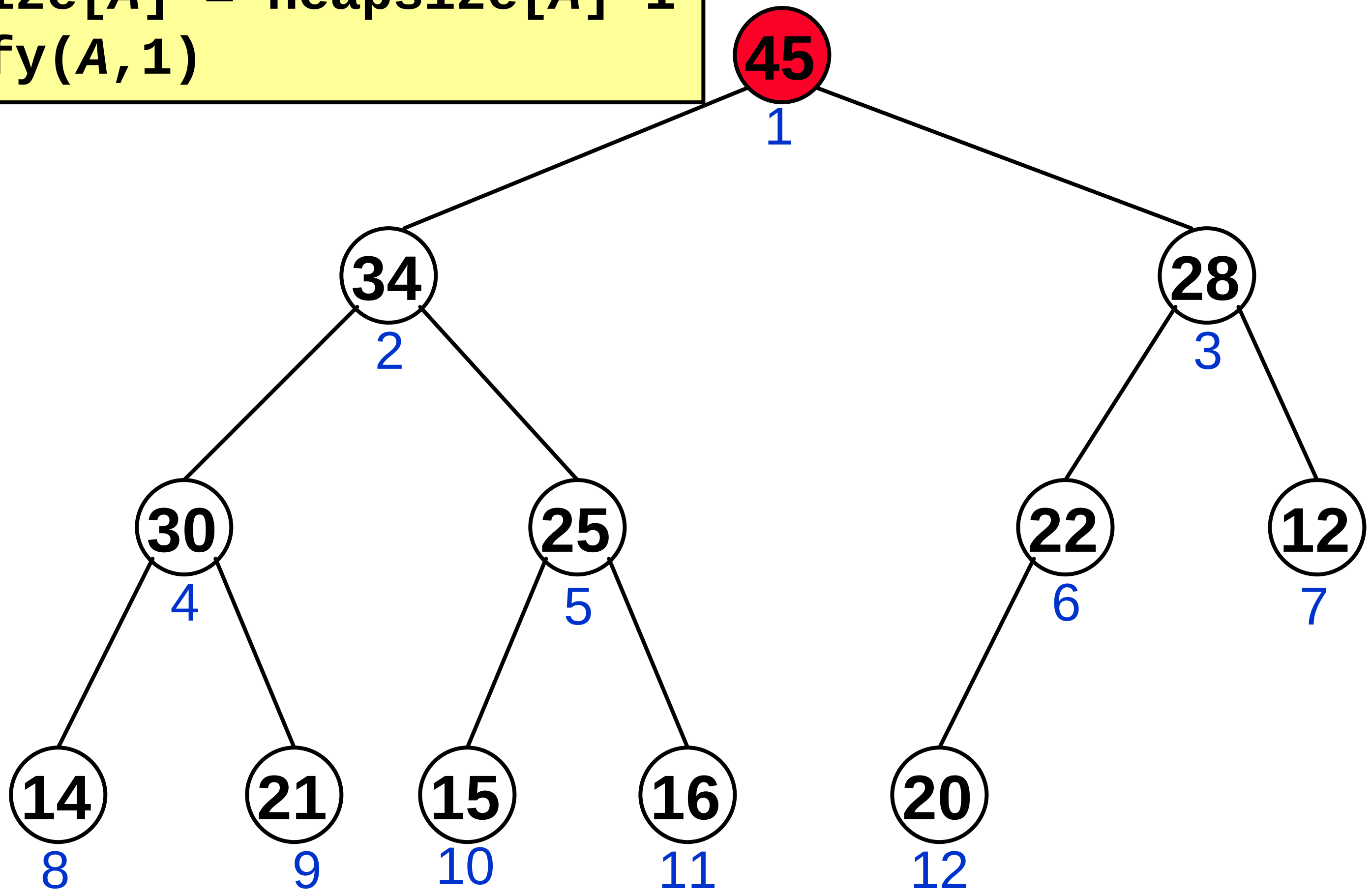


Heap Sort

```
Heap-Sort(A)
  …
  FOR i = length[A] DOWNTO 2
     DO "scambia A[1] e A[i]"
        heapsize[A] = heapsize[A]-1
        Heapify(A,1)
```

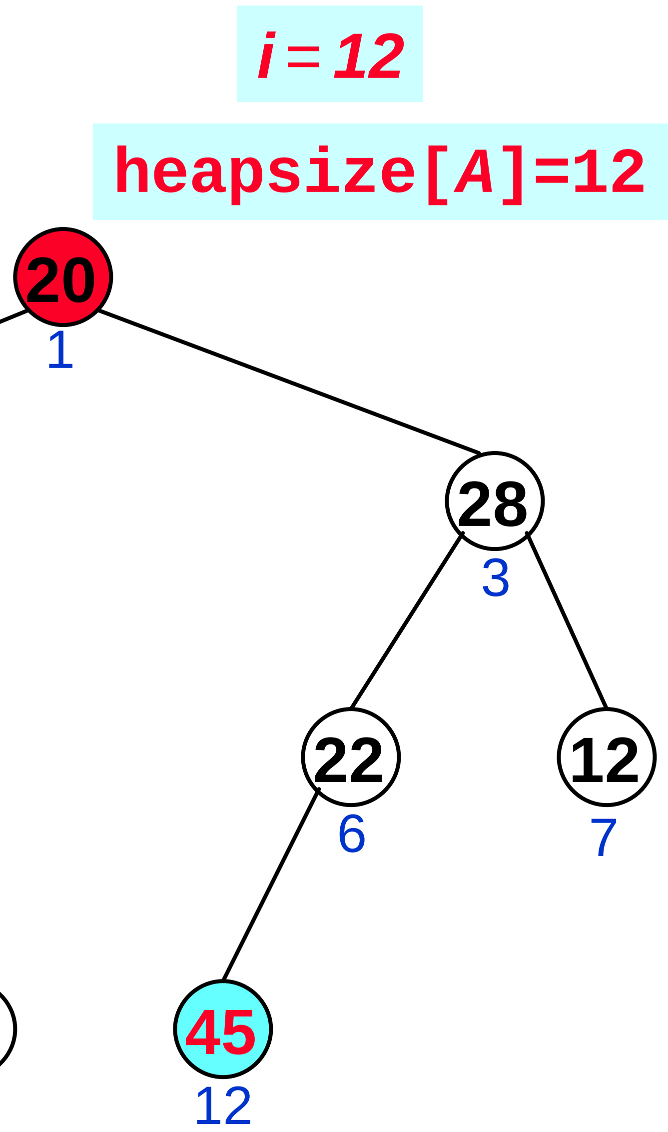
i = *12*

heapsize[*A*]=12



Heap Sort

```
Heap-Sort(A)
  …
  FOR i = length[A] DOWNTO 2
     DO "scambia A[1] e A[i]"
        heapsize[A] = heapsize[A]-1
        Heapify(A,1)
```

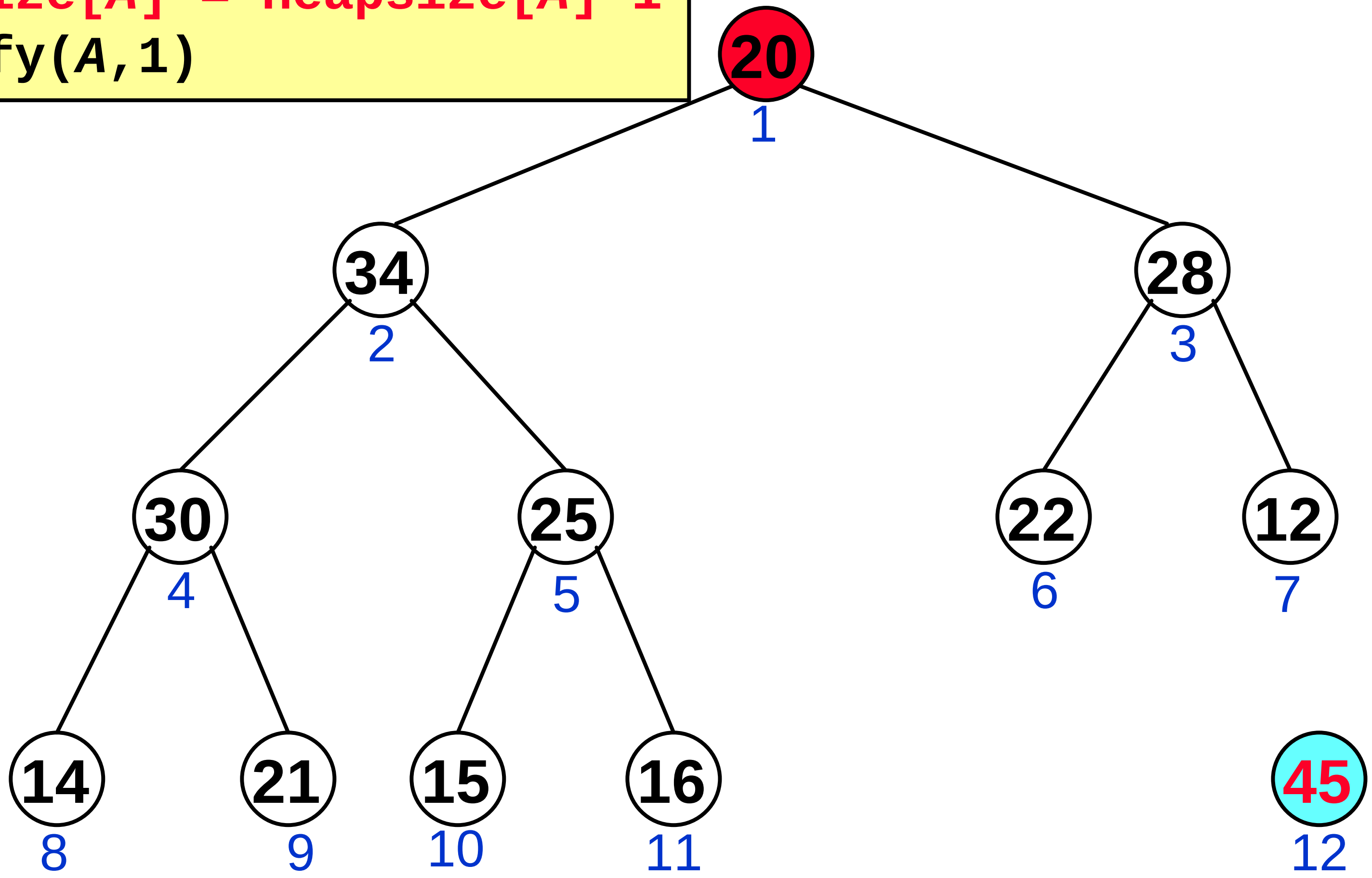


Heap Sort

```
Heap-Sort(A)
  …
  FOR i = length[A] DOWNTO 2
     DO "scambia A[1] e A[i]"
        heapsize[A] = heapsize[A]-1
        Heapify(A,1)
```

i = 12

heapsize[*A*]=11

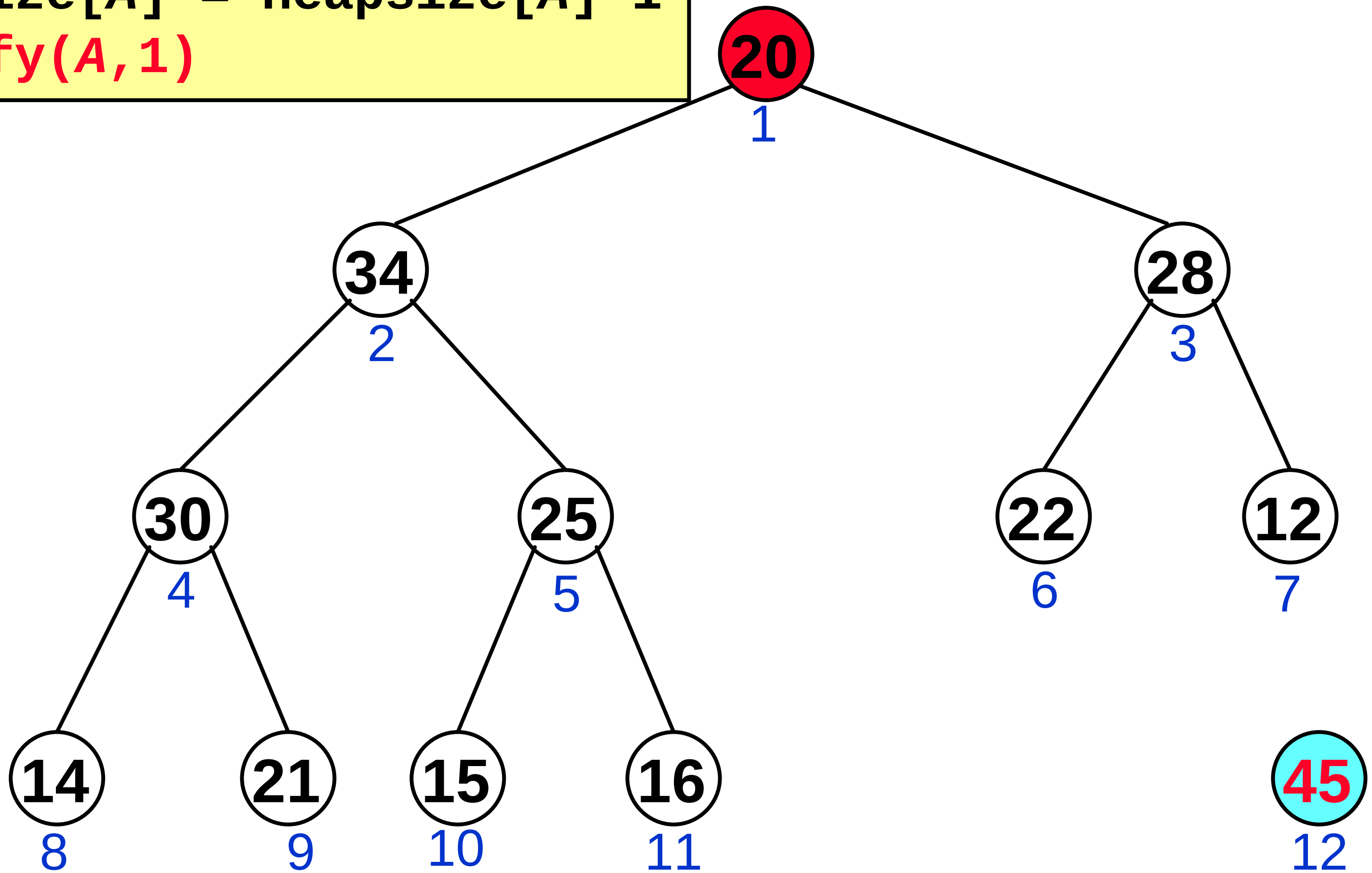


Heap Sort

```
Heap-Sort(A)
  …
  FOR i = length[A] DOWNTO 2
     DO “scambia A[1] e A[i]”
        heapsize[A] = heapsize[A]-1
        Heapify(A,1)
```

i = 12

heapsize[A]=11

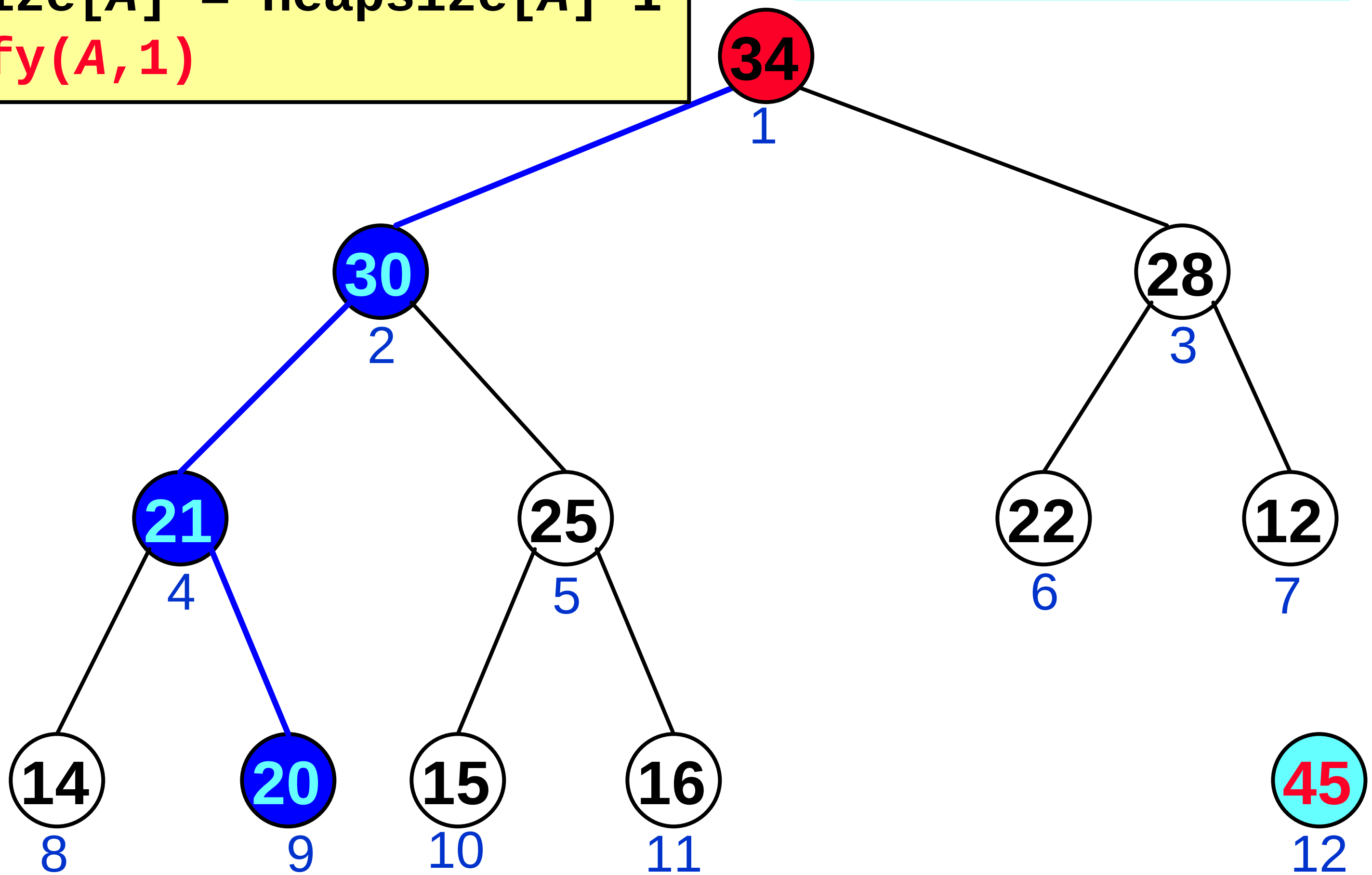


Heap Sort

```
Heap-Sort(A)
  …
  FOR i = length[A] DOWNTO 2
     DO “scambia A[1] e A[i]”
        heapsize[A] = heapsize[A]-1
        Heapify(A,1)
```

i = 12

heapsize[A]=11

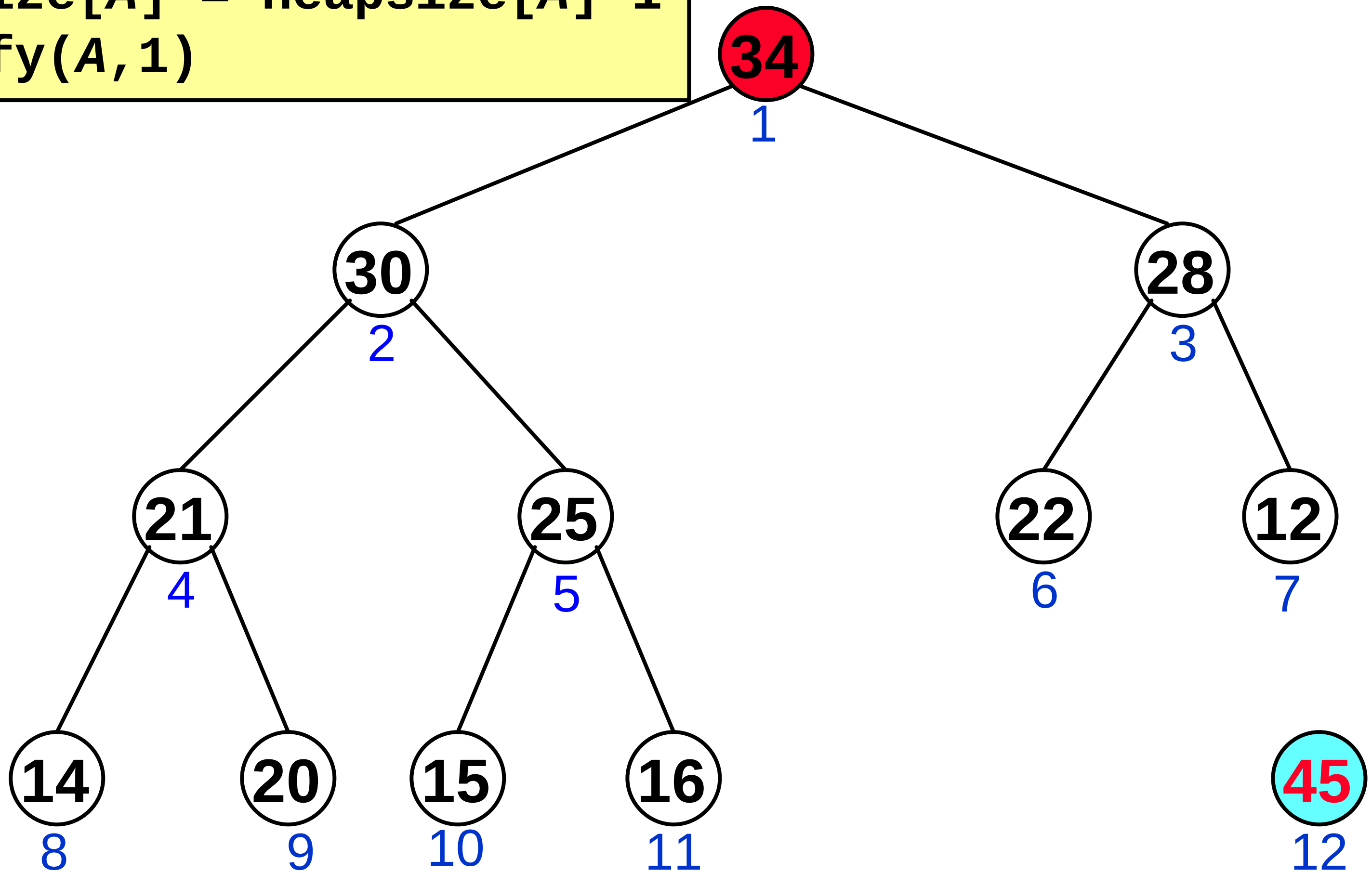


Heap Sort

```
Heap-Sort(A)
   …
   FOR i = length[A] DOWNTO 2
      DO "scambia A[1] e A[i]"
         heapsize[A] = heapsize[A]-1
         Heapify(A,1)
```

i = 11

heapsize[*A*]=11

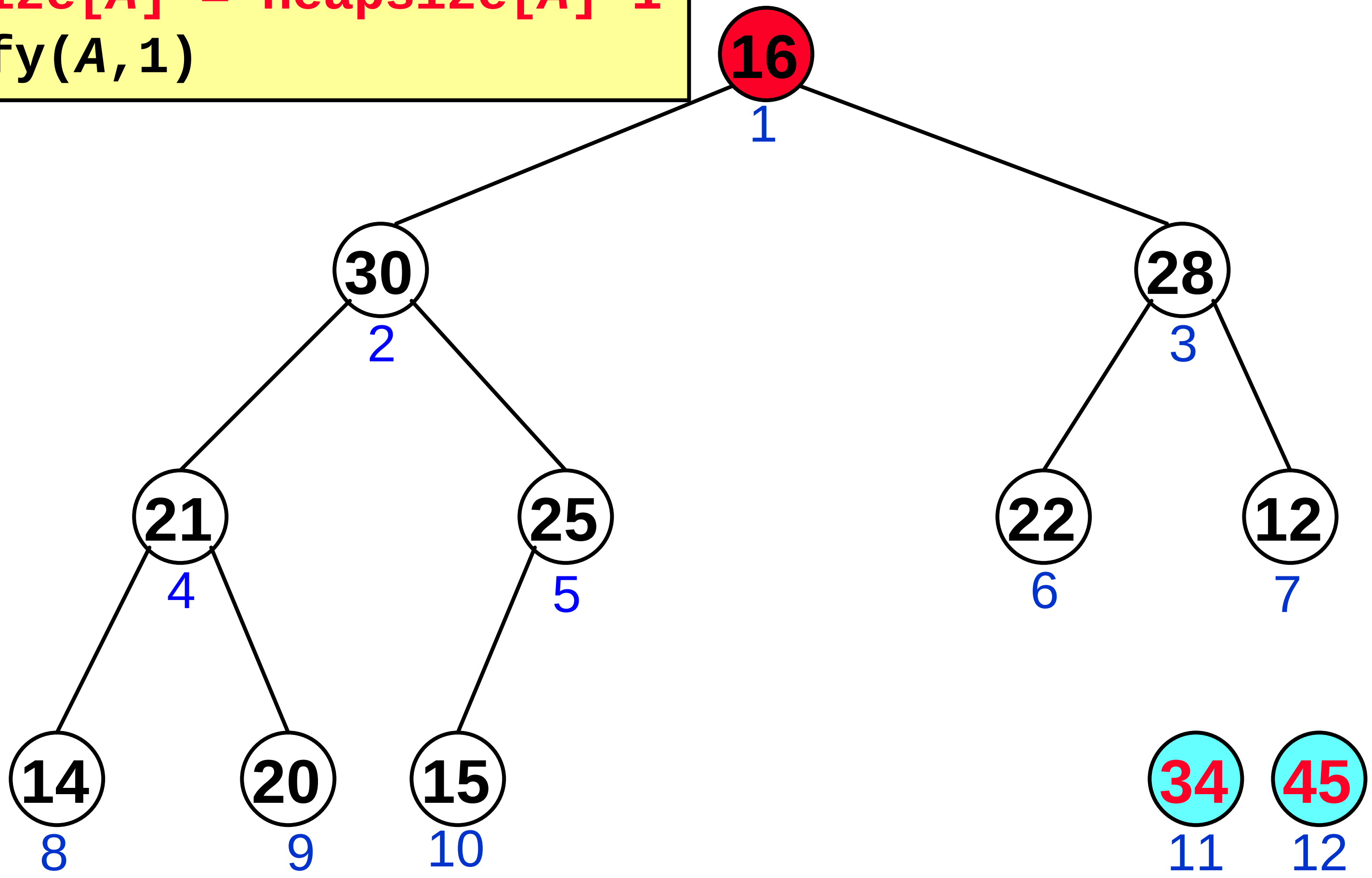


Heap Sort

```
Heap-Sort(A)
  …
  FOR i = length[A] DOWNTO 2
     DO "scambia A[1] e A[i]"
        heapsize[A] = heapsize[A]-1
        Heapify(A,1)
```

i = 11

heapsize[*A*]=10

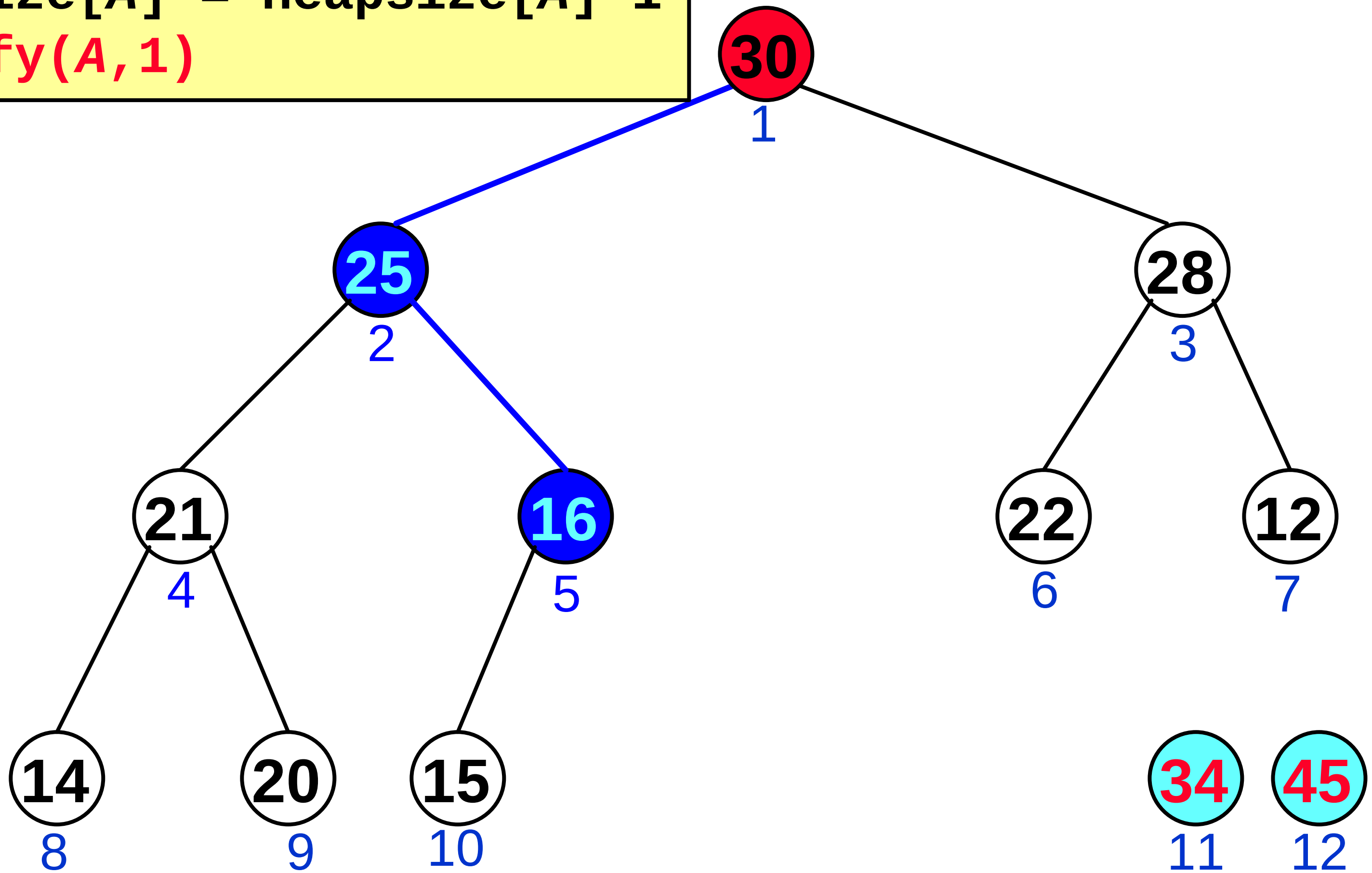


Heap Sort

```
Heap-Sort(A)
  …
  FOR i = length[A] DOWNTO 2
     DO “scambia A[1] e A[i]”
        heapsize[A] = heapsize[A]-1
        Heapify(A,1)
```

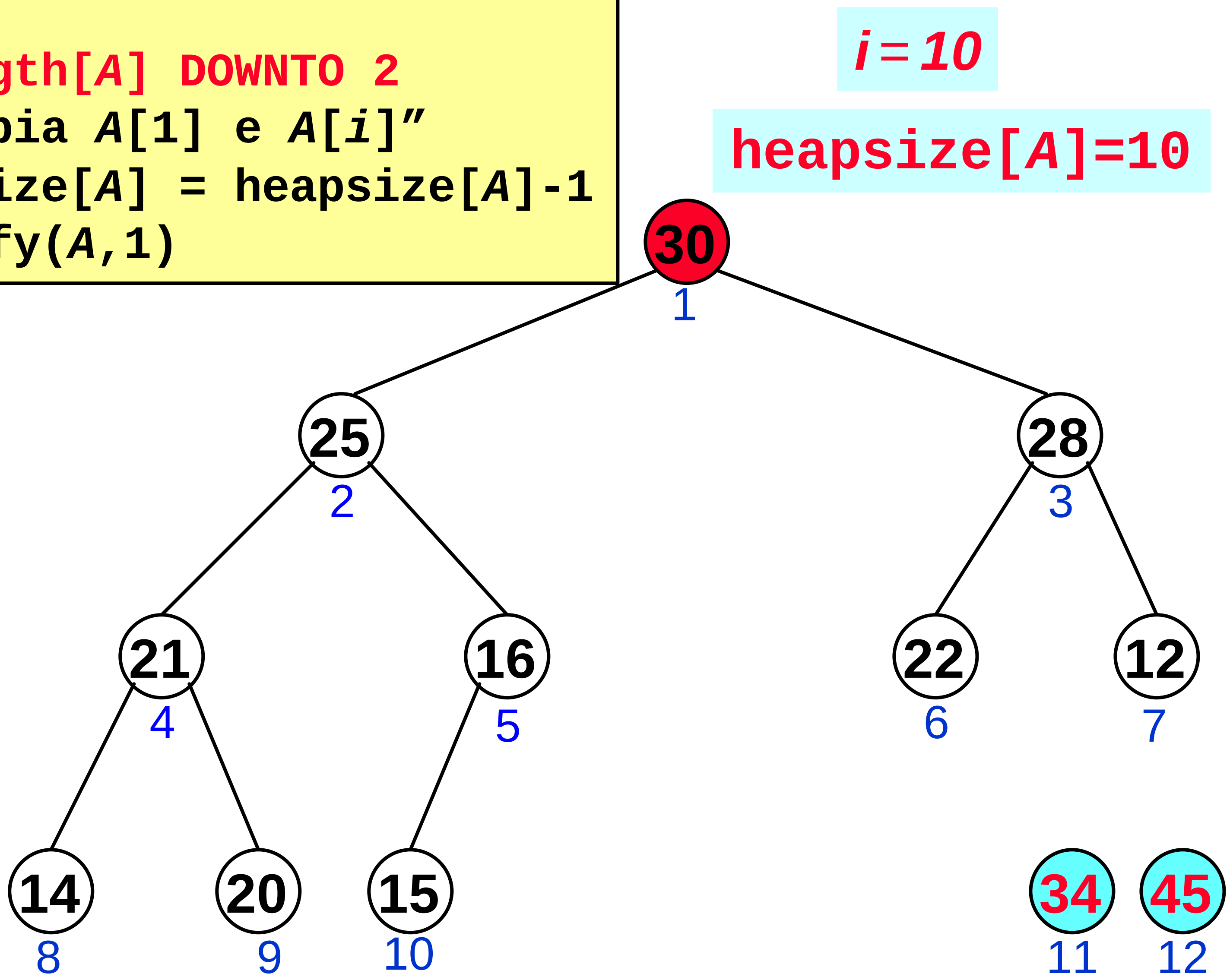
i = 11

heapsize[*A*]=10



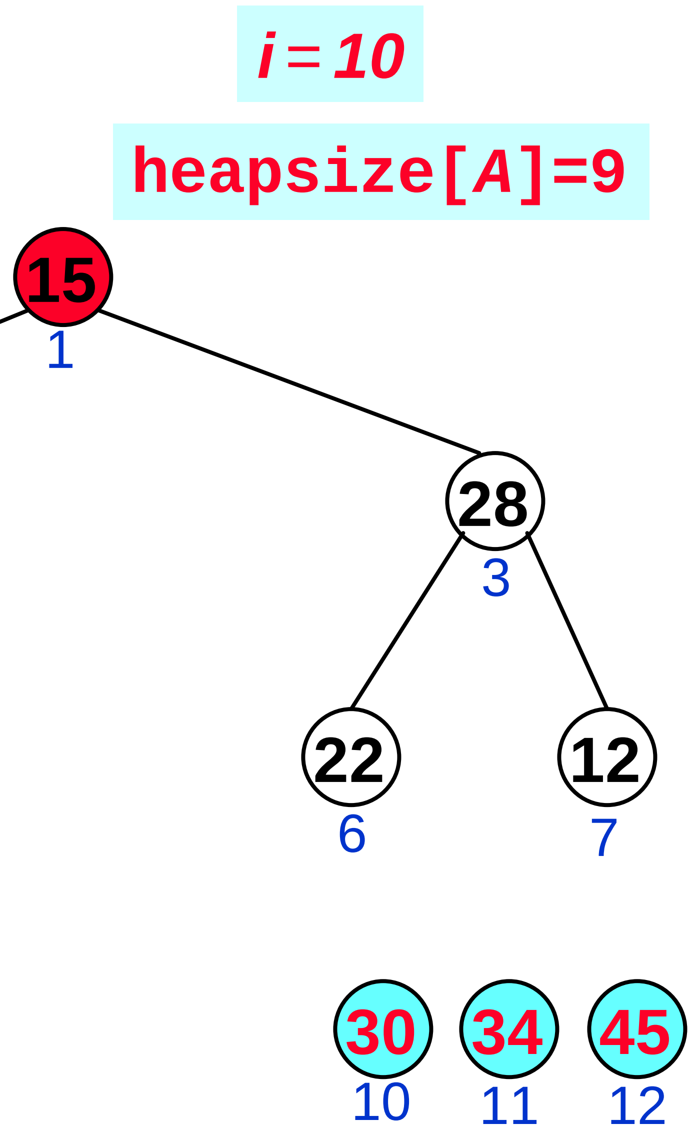
Heap Sort

```
Heap-Sort(A)
  …
  FOR i = length[A] DOWNTO 2
     DO "scambia A[1] e A[i]"
        heapsize[A] = heapsize[A]-1
        Heapify(A,1)
```



Heap Sort

```
Heap-Sort(A)
  …
  FOR i = length[A] DOWNTO 2
     DO "scambia A[1] e A[i]"
        heapsize[A] = heapsize[A]-1
        Heapify(A,1)
```

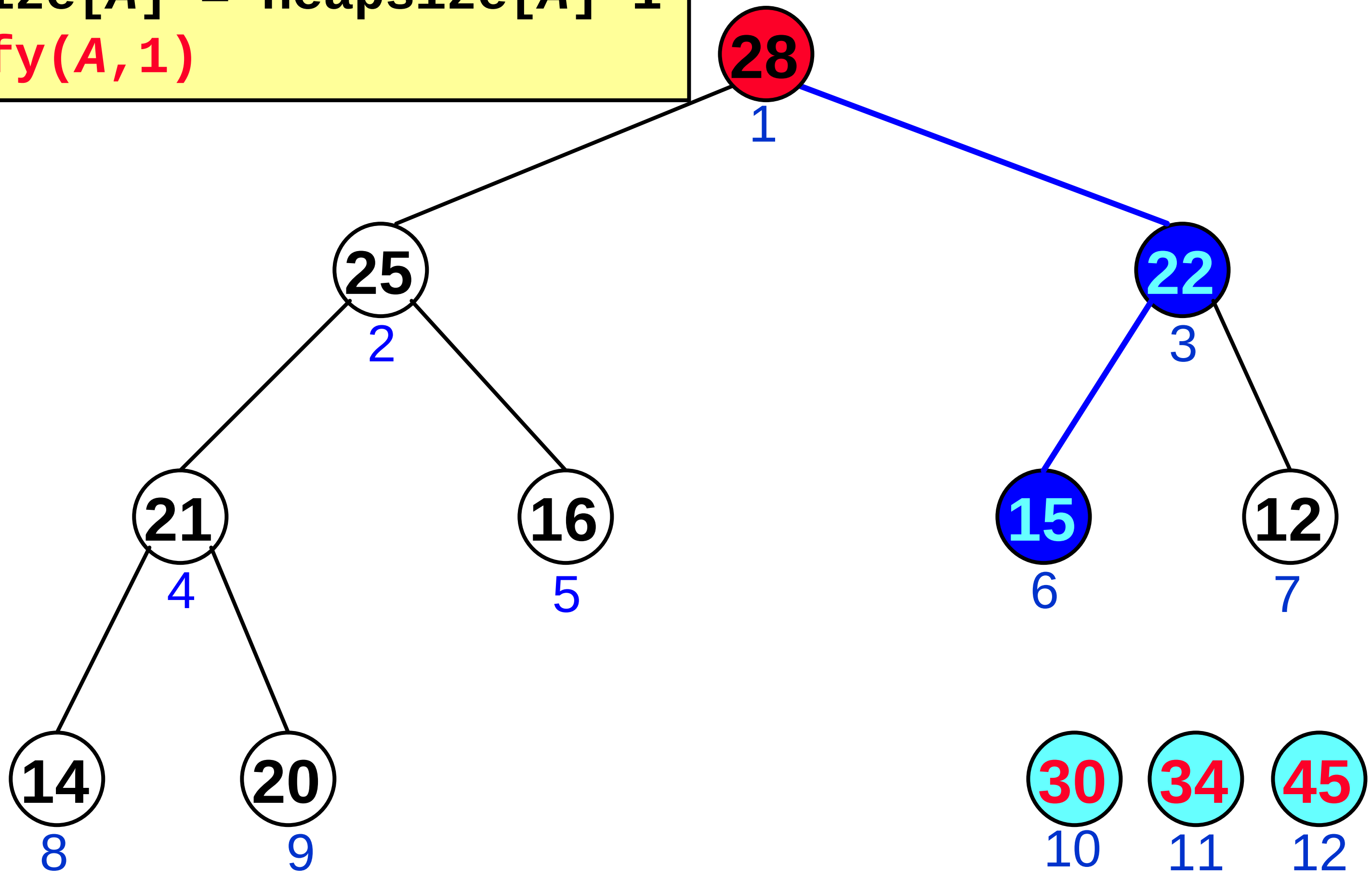


Heap Sort

```
Heap-Sort(A)
  …
  FOR i = length[A] DOWNTO 2
     DO "scambia A[1] e A[i]"
        heapsize[A] = heapsize[A]-1
        Heapify(A,1)
```

i = 10

heapsize[*A*]=9

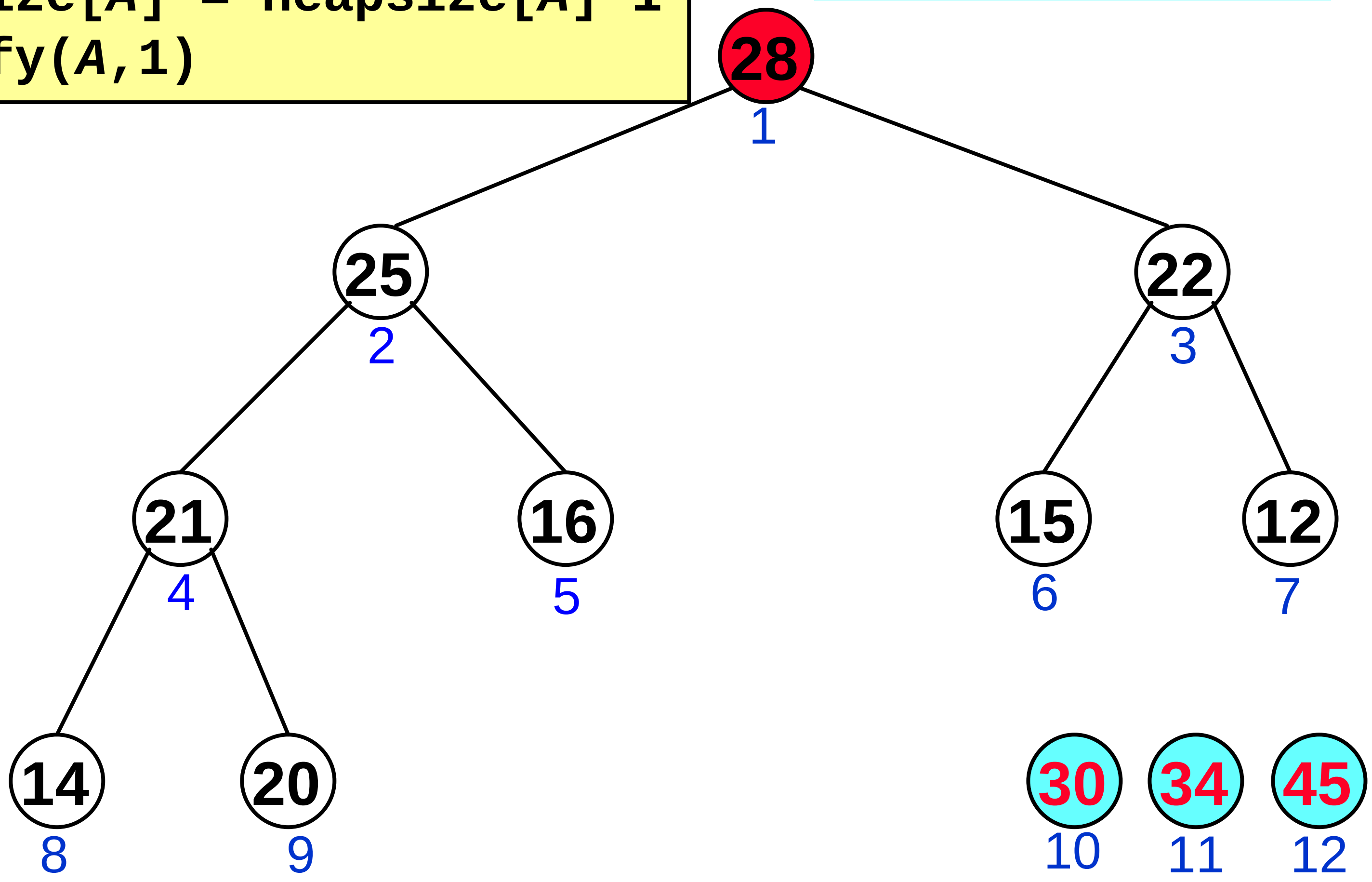


Heap Sort

```
Heap-Sort(A)
  …
  FOR i = length[A] DOWNTO 2
     DO "scambia A[1] e A[i]"
        heapsize[A] = heapsize[A]-1
        Heapify(A,1)
```

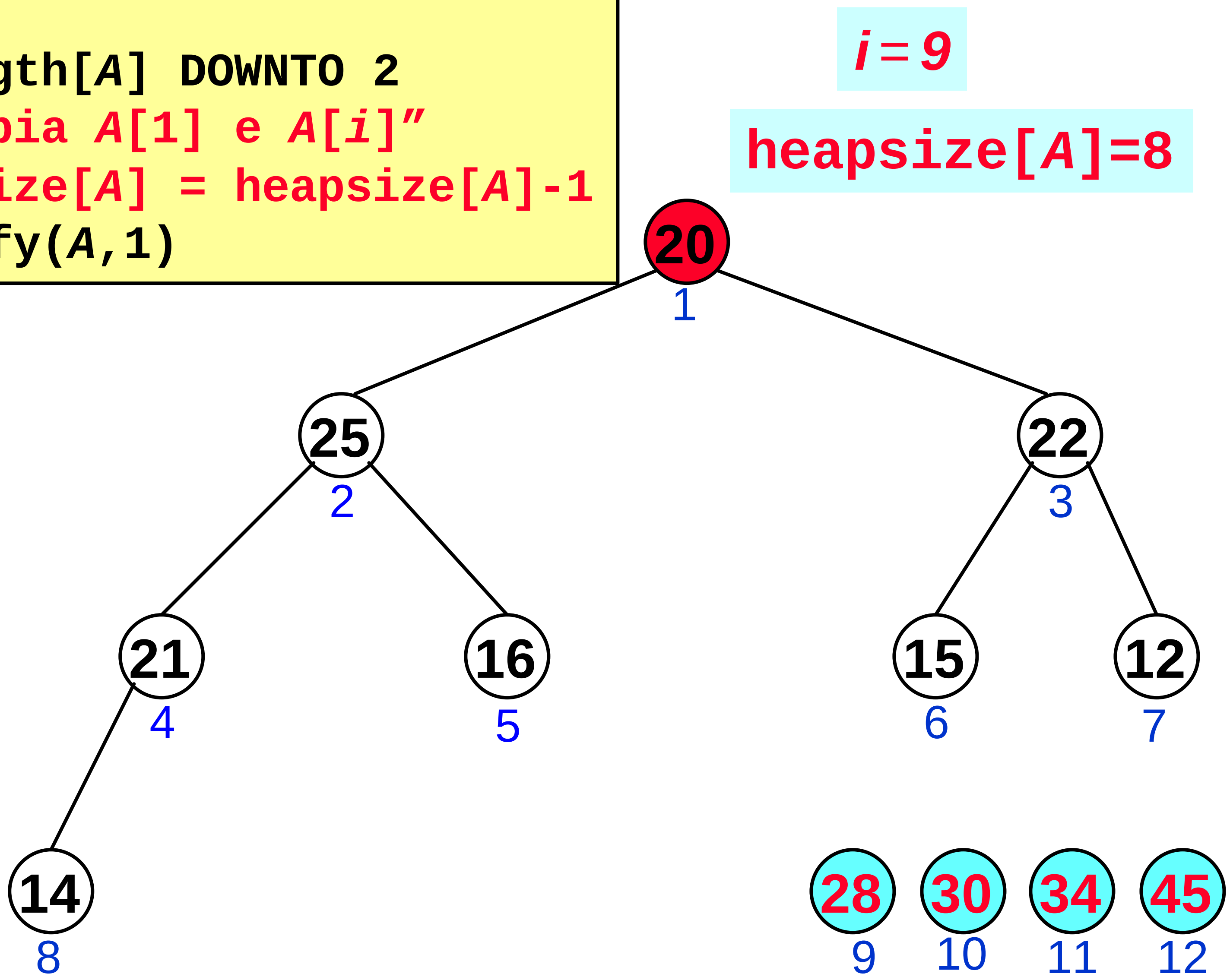
i = *9*

heapsize[*A*]=9



Heap Sort

```
Heap-Sort(A)
  …
  FOR i = length[A] DOWNTO 2
     DO "scambia A[1] e A[i]"
        heapsize[A] = heapsize[A]-1
        Heapify(A,1)
```

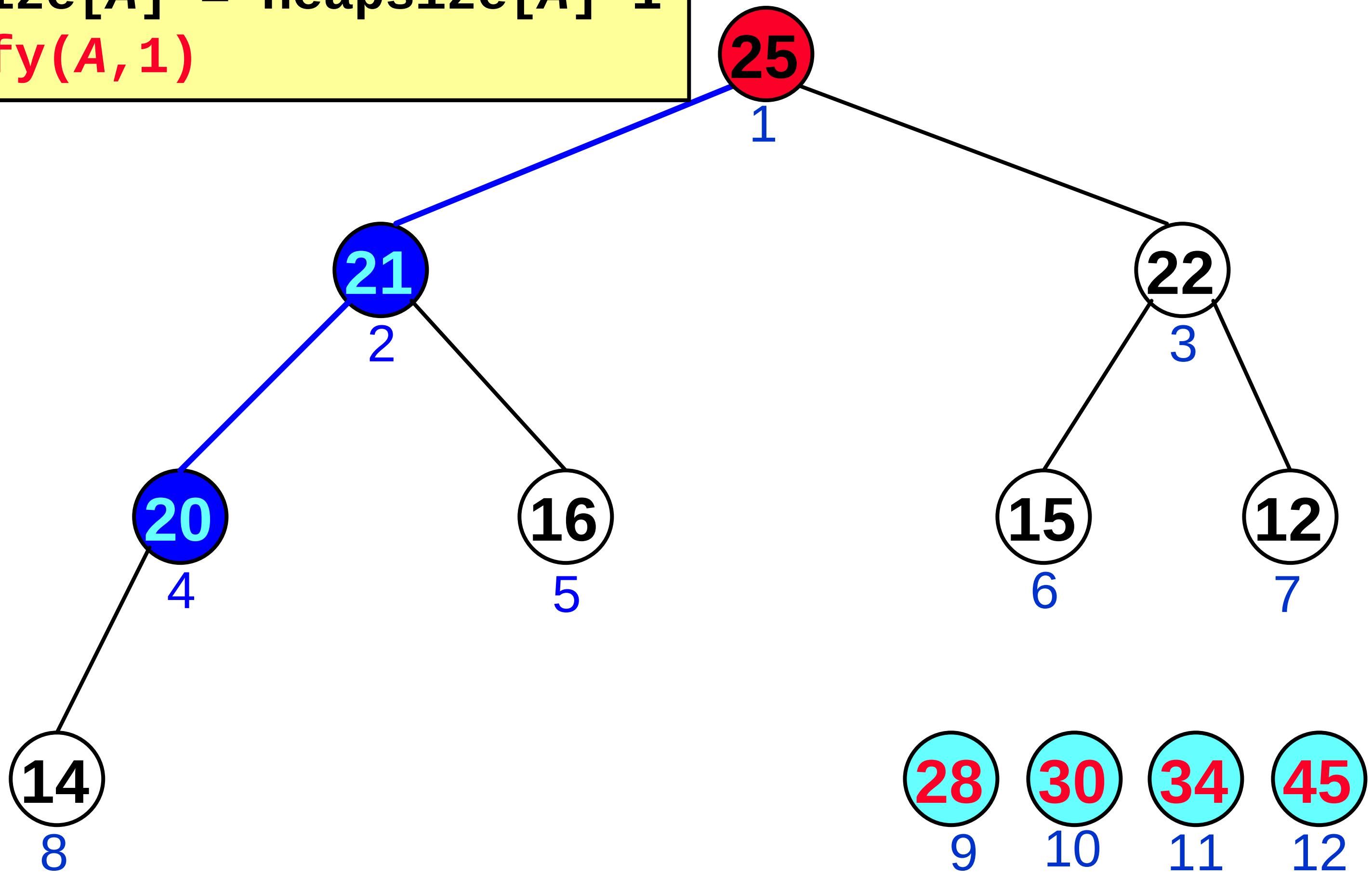


Heap Sort

```
Heap-Sort(A)
  …
  FOR i = length[A] DOWNTO 2
     DO "scambia A[1] e A[i]"
        heapsize[A] = heapsize[A]-1
        Heapify(A,1)
```

i = 9

heapsize[*A*]=8

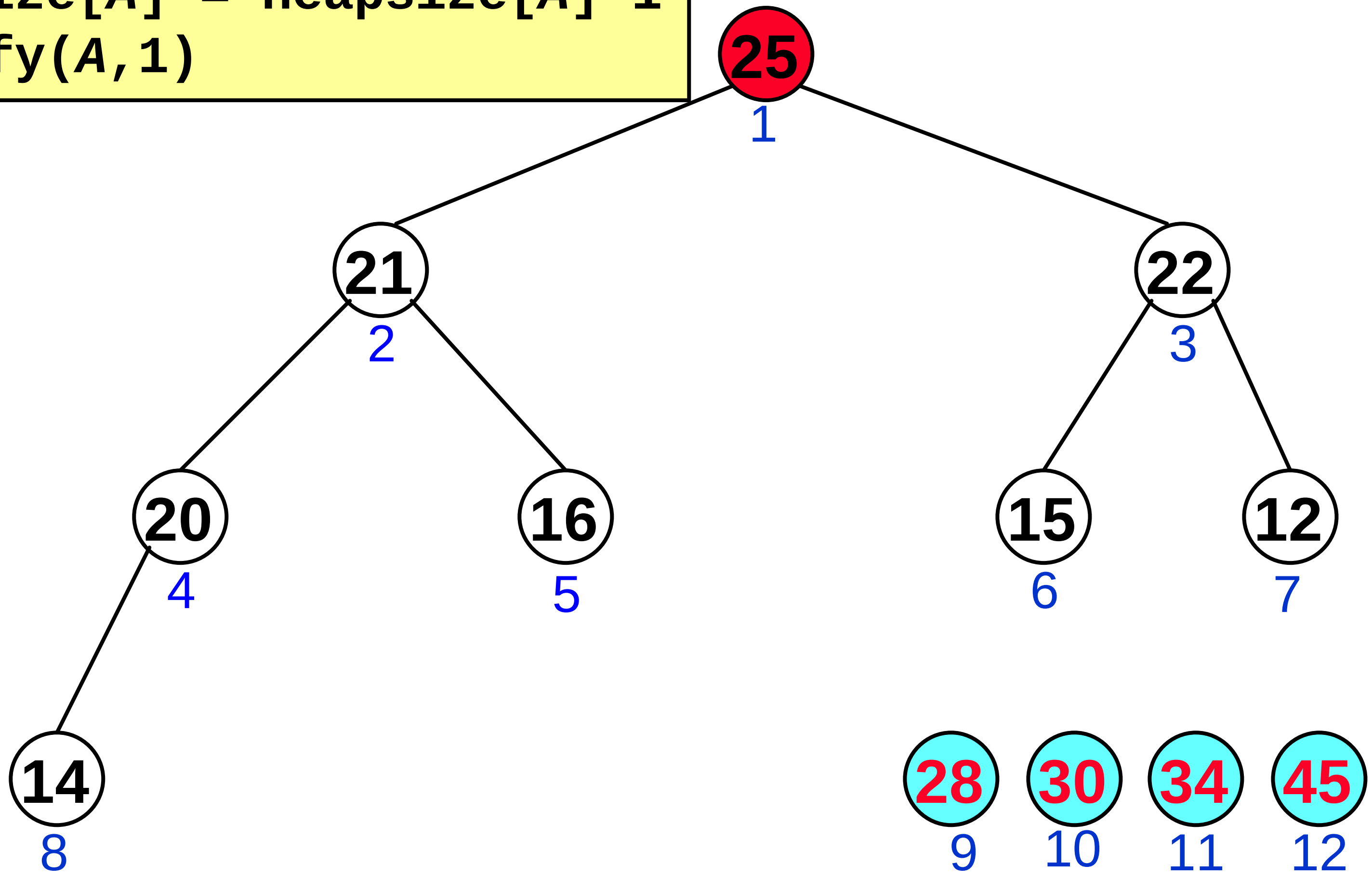


Heap Sort

```
Heap-Sort(A)
  …
  FOR i = length[A] DOWNTO 2
     DO “scambia A[1] e A[i]”
        heapsize[A] = heapsize[A]-1
        Heapify(A,1)
```

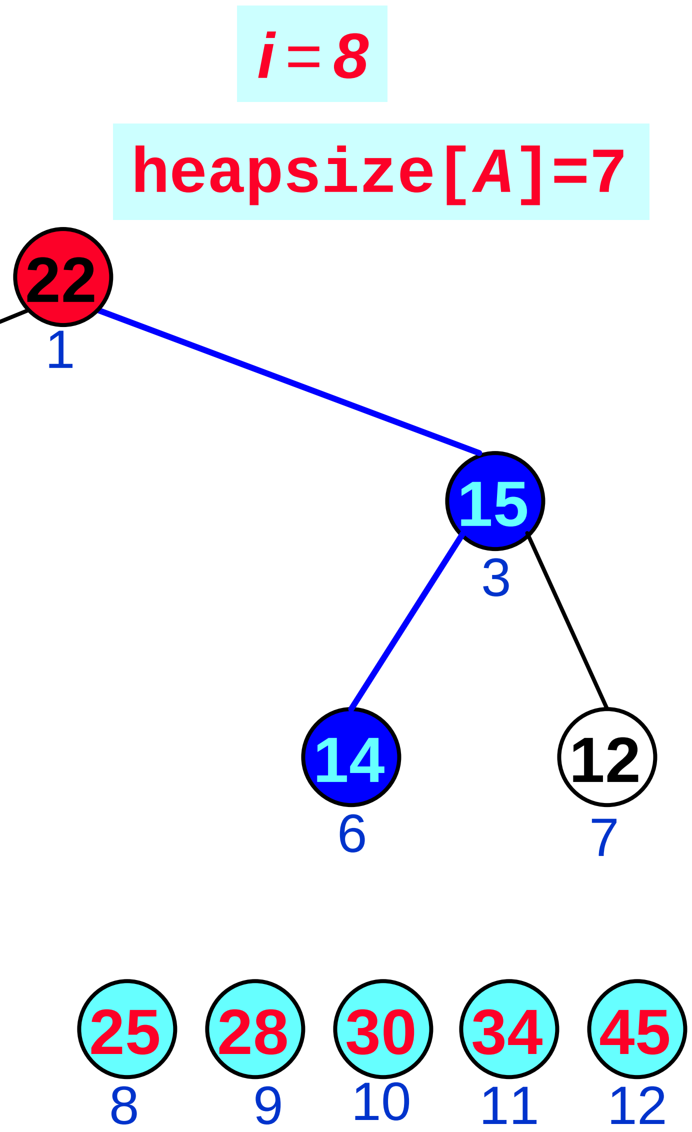
$i = 8$

heapsize[*A*]=8



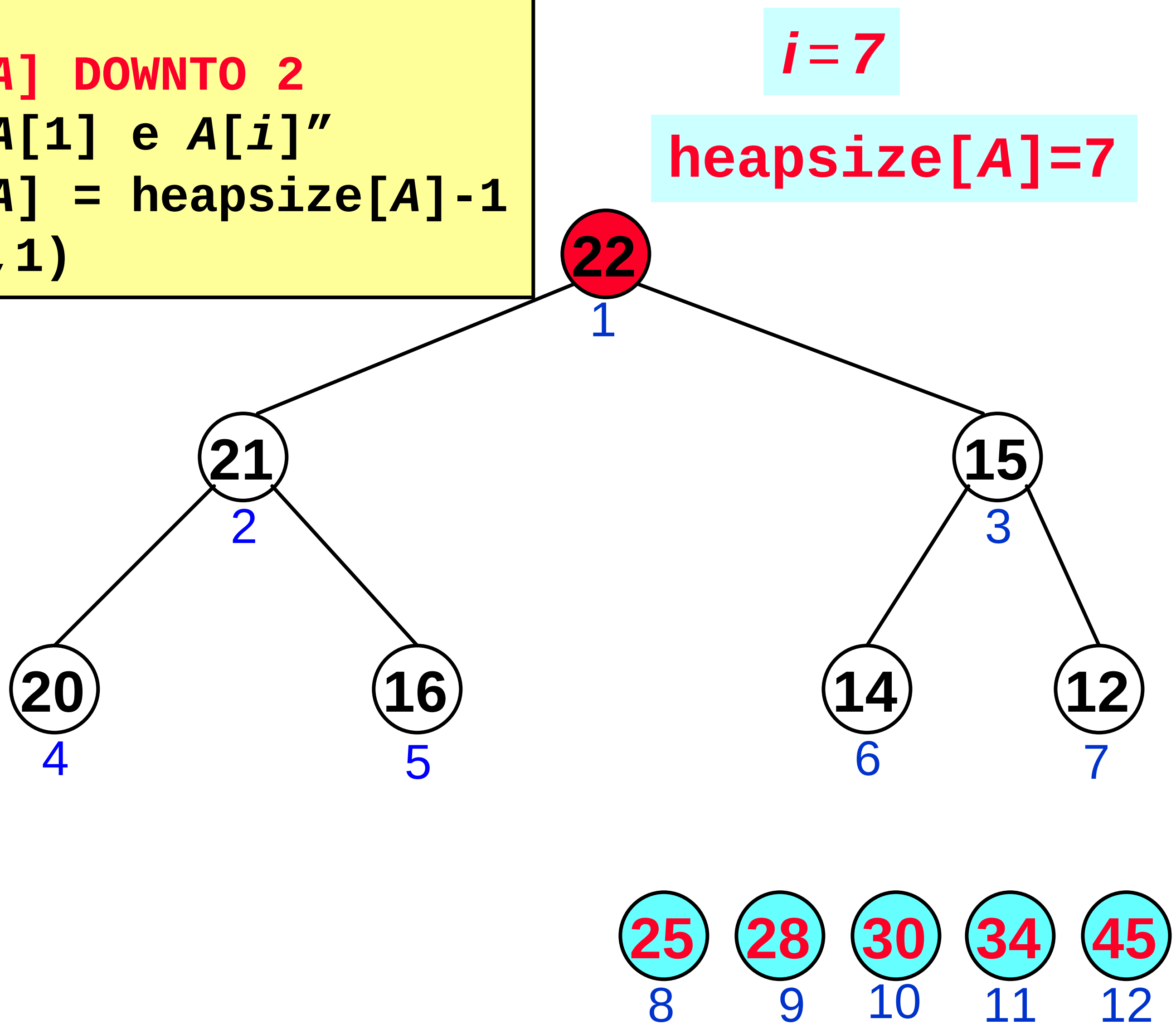
Heap Sort

```
Heap-Sort(A)
  …
  FOR i = length[A] DOWNTO 2
     DO "scambia A[1] e A[i]"
        heapsize[A] = heapsize[A]-1
        Heapify(A,1)
```



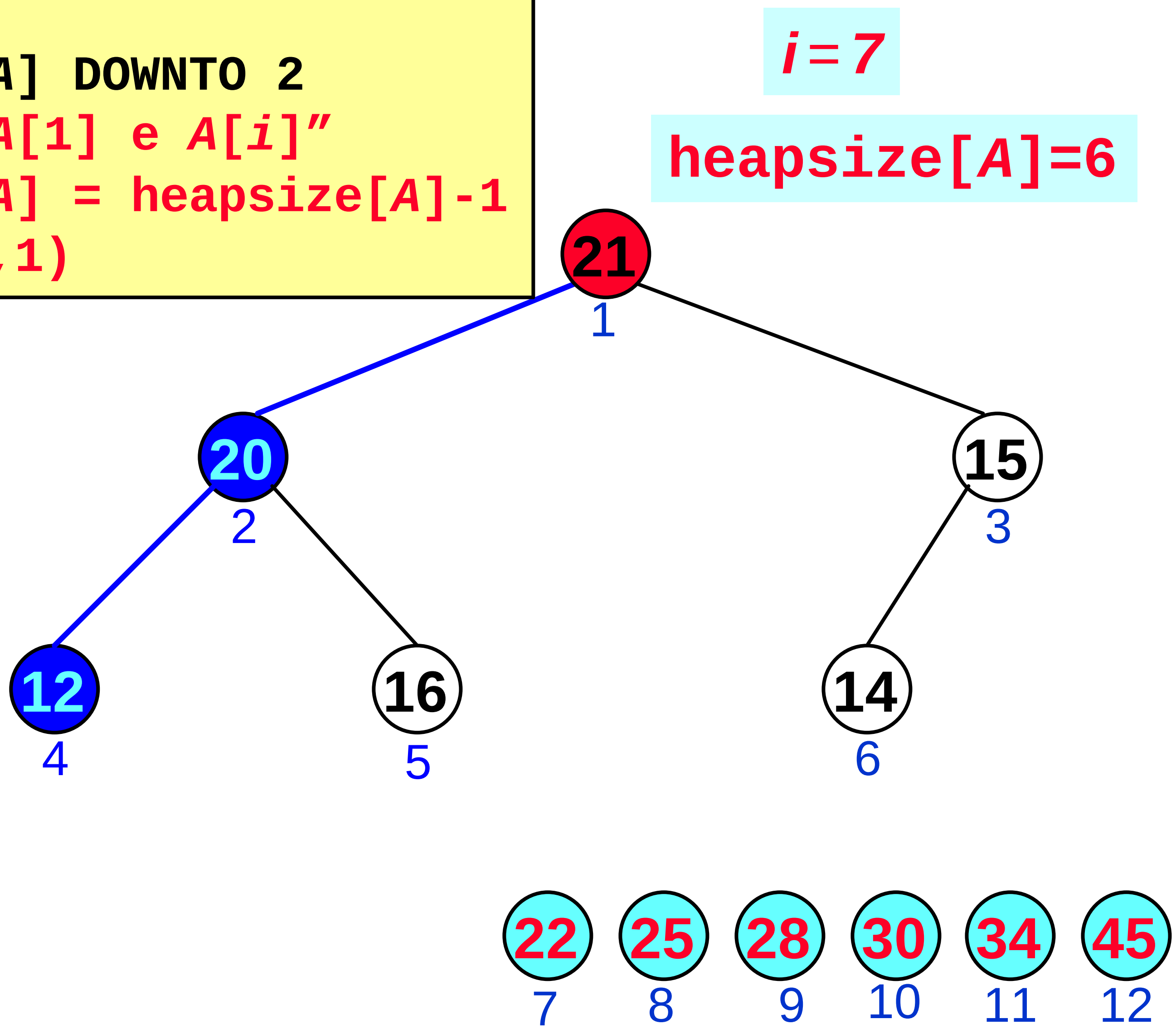
Heap Sort

```
Heap-Sort(A)
  …
  FOR i = length[A] DOWNTO 2
     DO "scambia A[1] e A[i]"
        heapsize[A] = heapsize[A]-1
        Heapify(A,1)
```



Heap Sort

```
Heap-Sort(A)
  …
  FOR i = length[A] DOWNTO 2
     DO "scambia A[1] e A[i]"
        heapsize[A] = heapsize[A]-1
        Heapify(A,1)
```

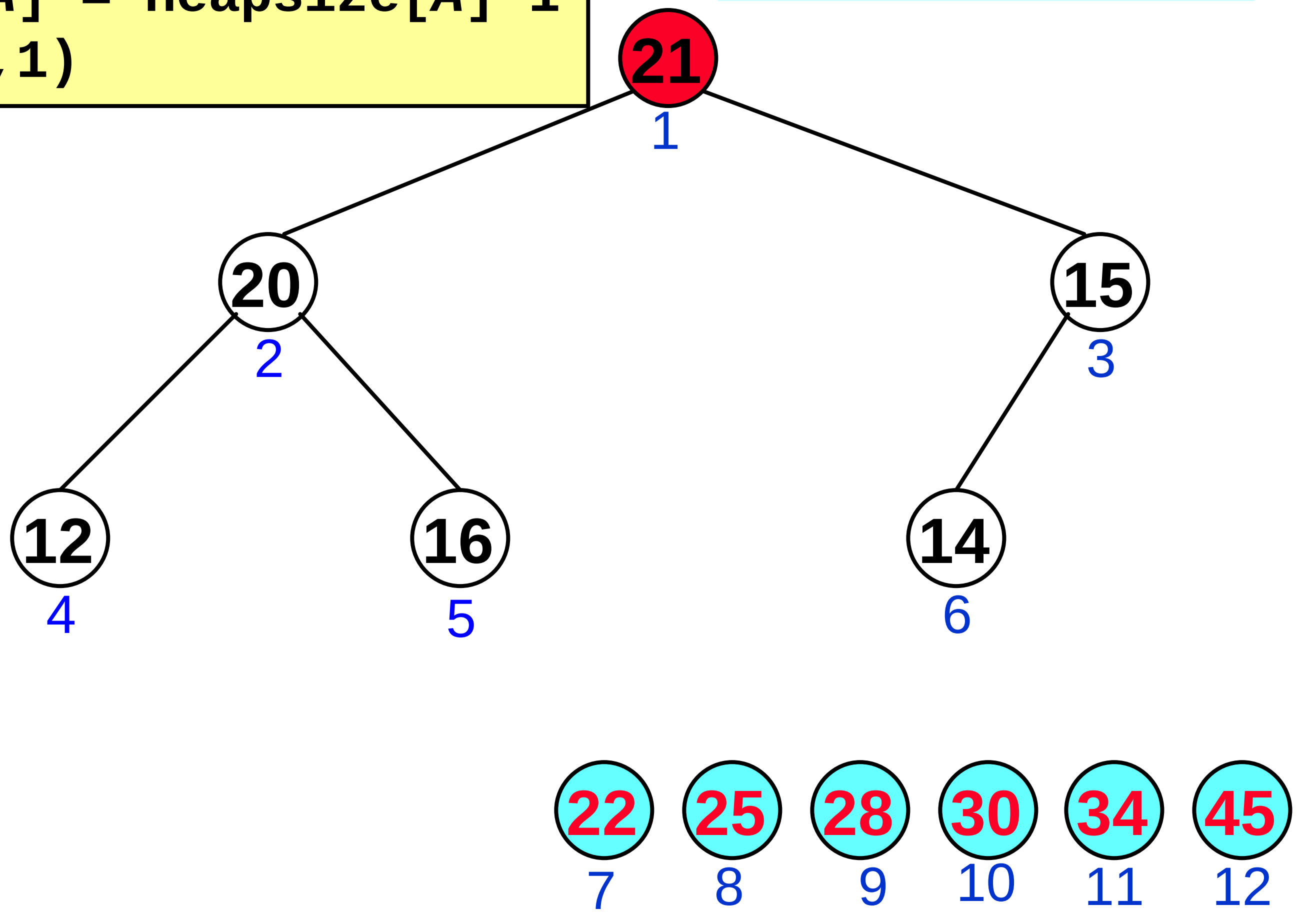


Heap Sort

```
Heap-Sort(A)
  …
  FOR i = length[A] DOWNTO 2
     DO "scambia A[1] e A[i]"
        heapsize[A] = heapsize[A]-1
        Heapify(A,1)
```

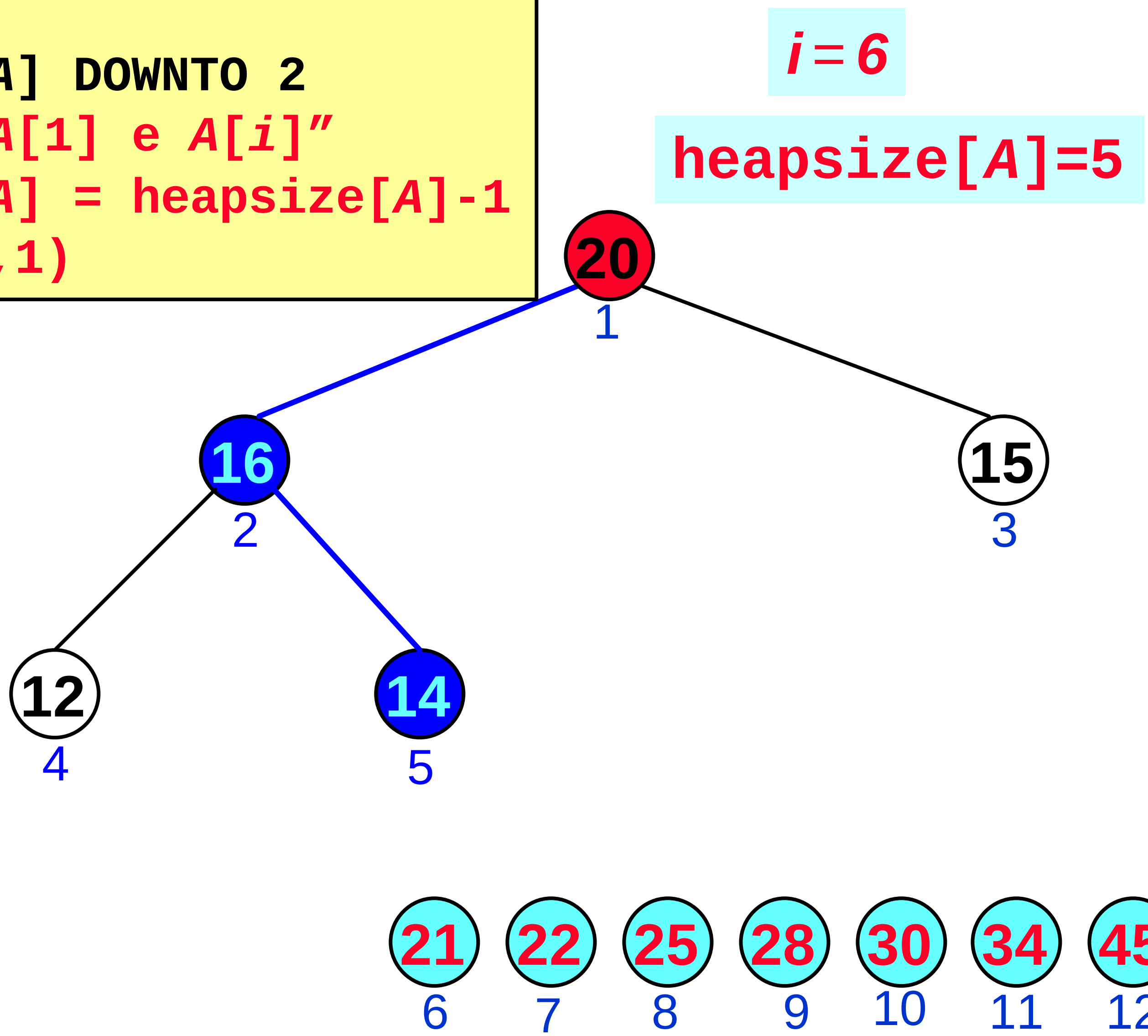
i = 6

heapsize[*A*]=6



Heap Sort

```
Heap-Sort(A)
  …
  FOR i = length[A] DOWNTO 2
     DO "scambia A[1] e A[i]"
        heapsize[A] = heapsize[A]-1
        Heapify(A,1)
```

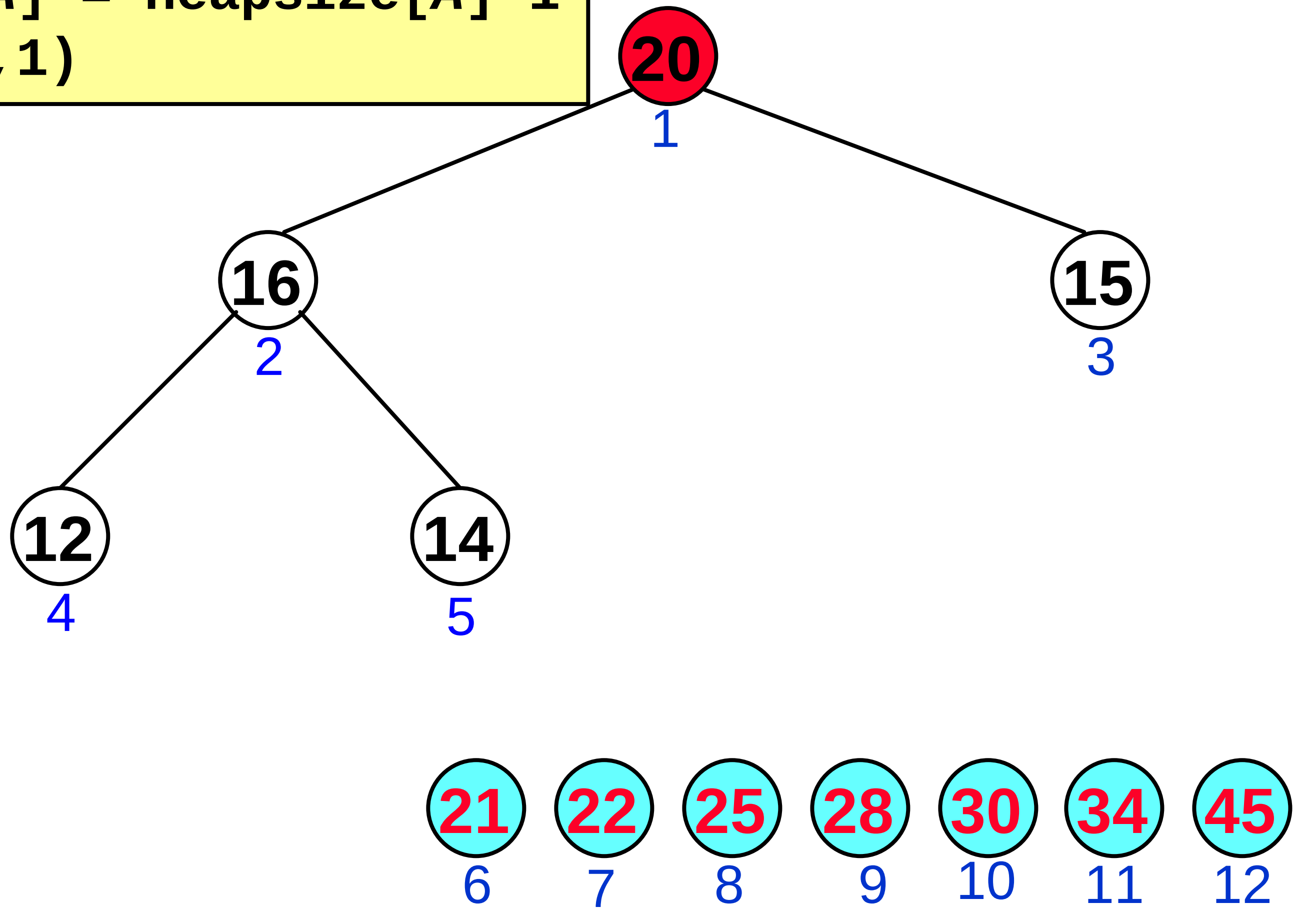


Heap Sort

```
Heap-Sort(A)
  …
  FOR i = length[A] DOWNTO 2
     DO "scambia A[1] e A[i]"
        heapsize[A] = heapsize[A]-1
        Heapify(A,1)
```

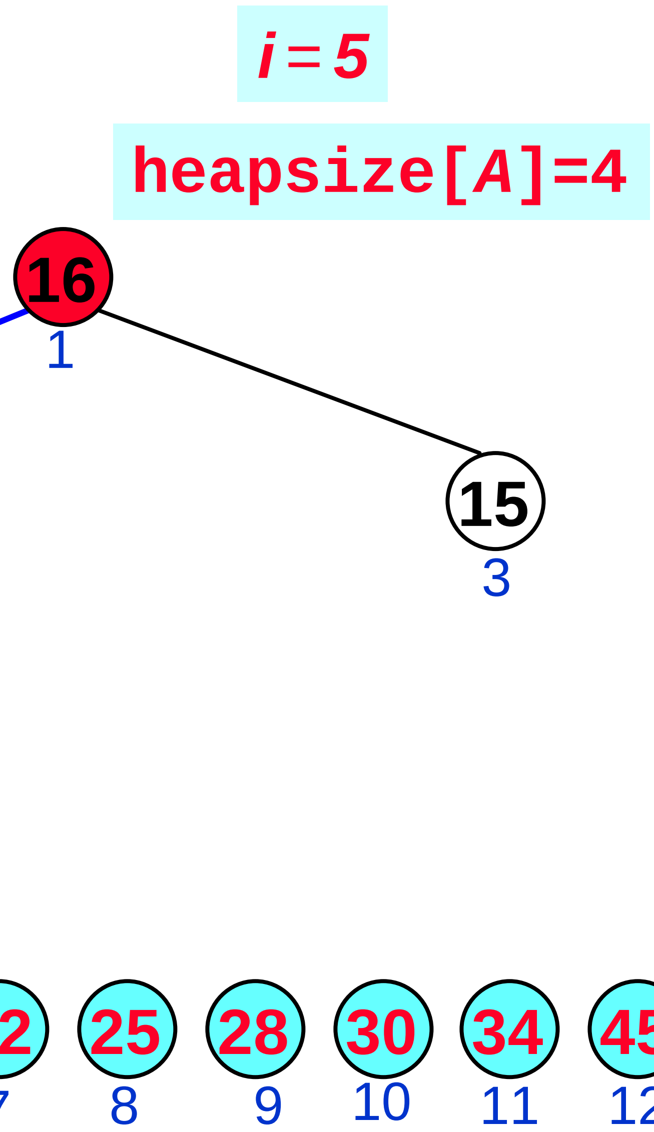
i = 5

heapsize[*A*]=5



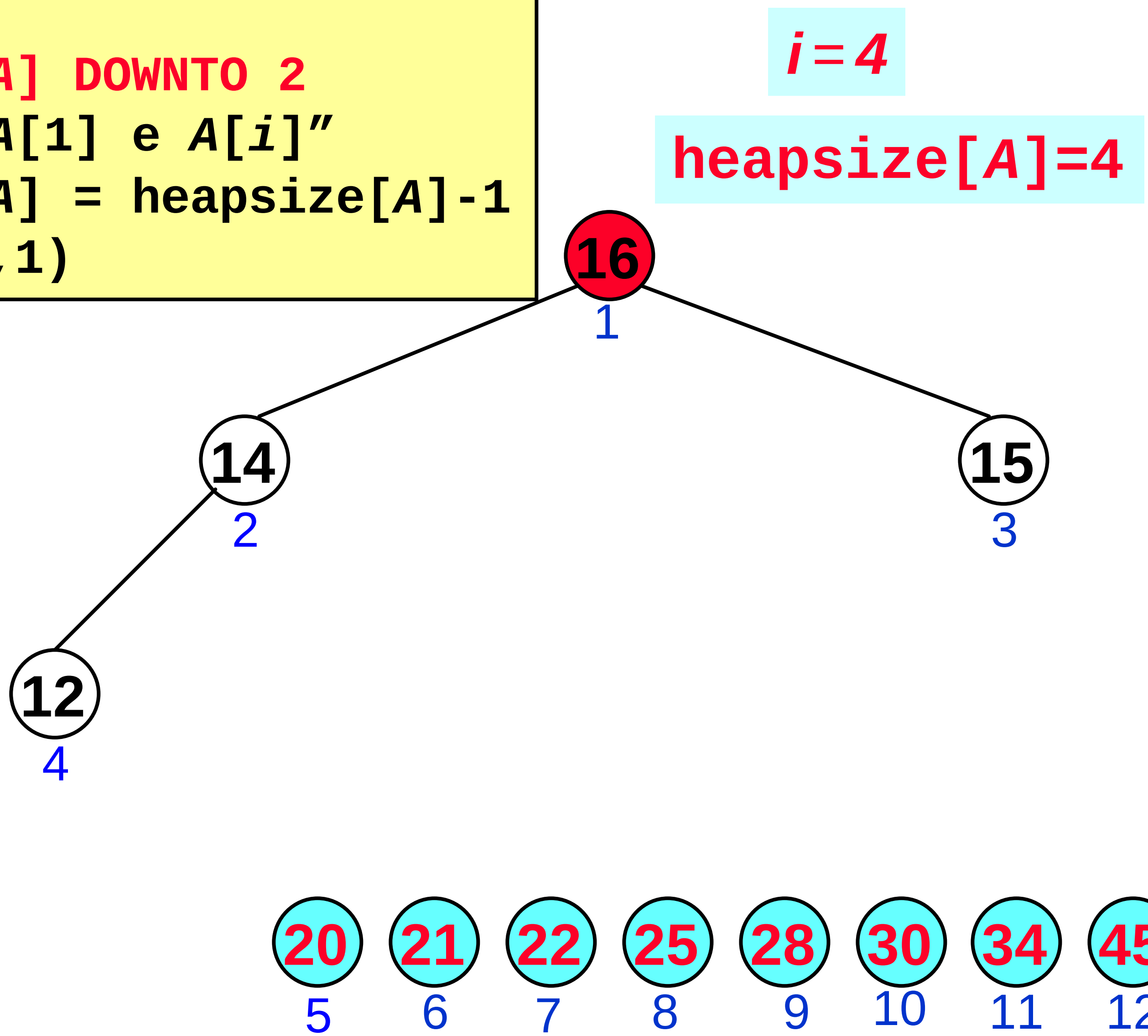
Heap Sort

```
Heap-Sort(A)
  …
  FOR i = length[A] DOWNTO 2
     DO "scambia A[1] e A[i]"
        heapsize[A] = heapsize[A]-1
        Heapify(A,1)
```



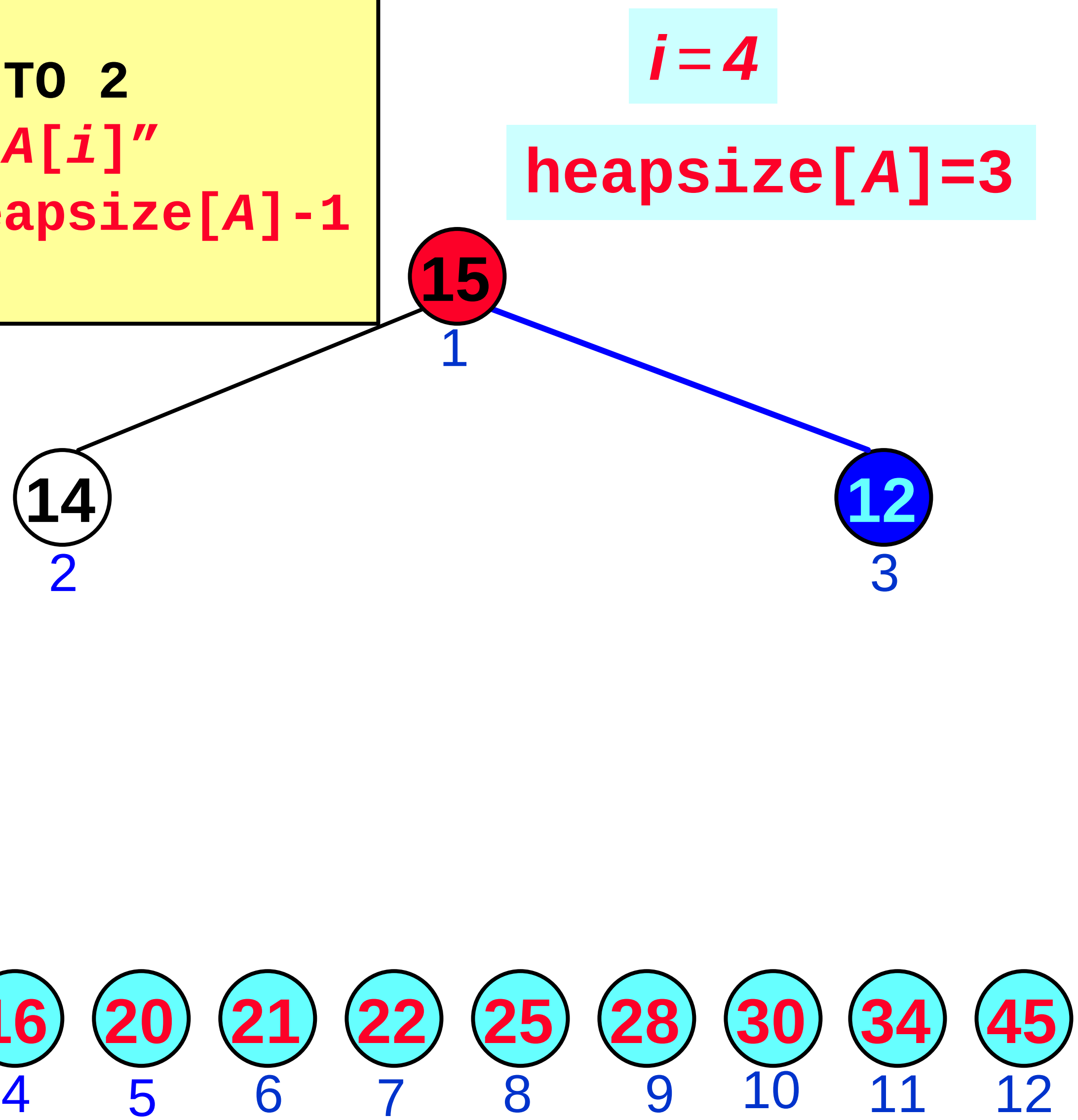
Heap Sort

```
Heap-Sort(A)
  …
  FOR i = length[A] DOWNTO 2
     DO "scambia A[1] e A[i]"
        heapsize[A] = heapsize[A]-1
        Heapify(A,1)
```



Heap Sort

```
Heap-Sort(A)
  …
  FOR i = length[A] DOWNTO 2
     DO "scambia A[1] e A[i]"
        heapsize[A] = heapsize[A]-1
        Heapify(A,1)
```

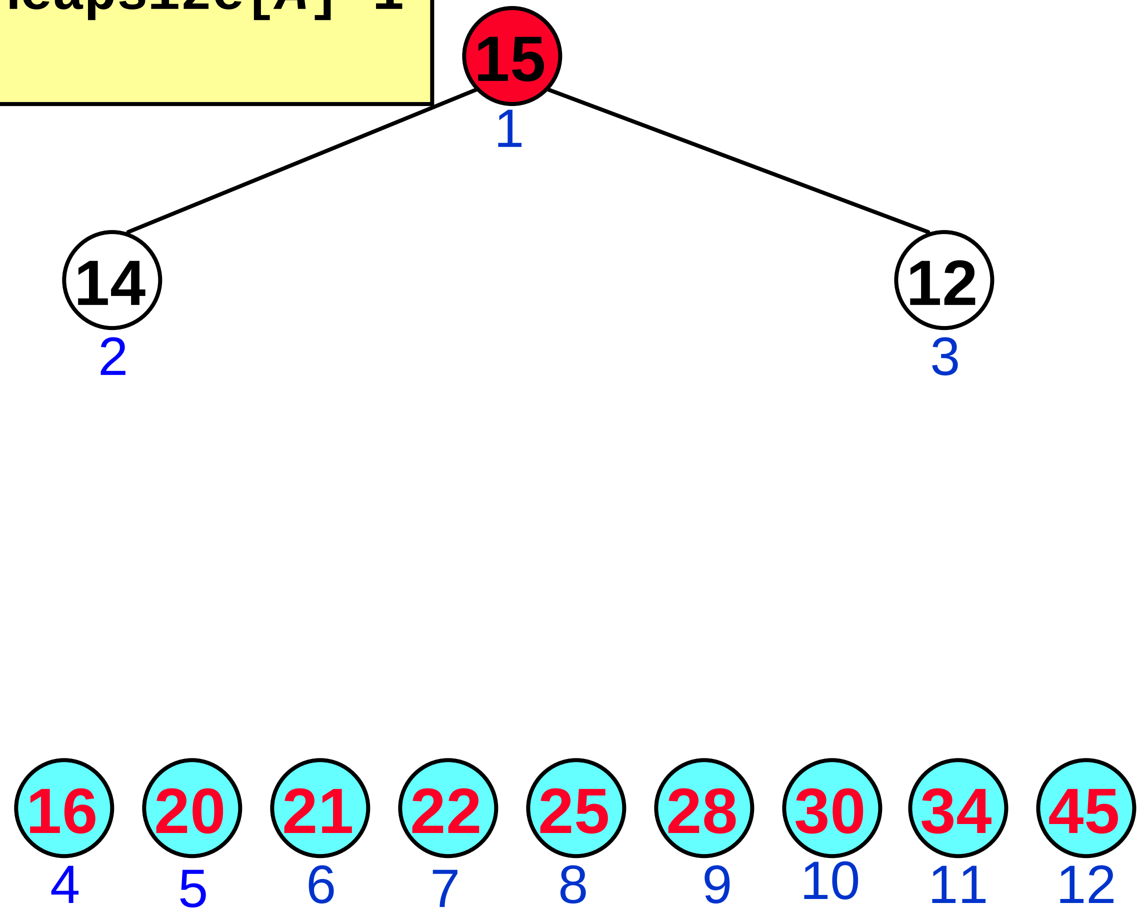


Heap Sort

```
Heap-Sort(A)
  …
  FOR i = length[A] DOWNTO 2
     DO "scambia A[1] e A[i]"
        heapsize[A] = heapsize[A]-1
        Heapify(A,1)
```

***i* = 3**

heapsize[*A*]=3

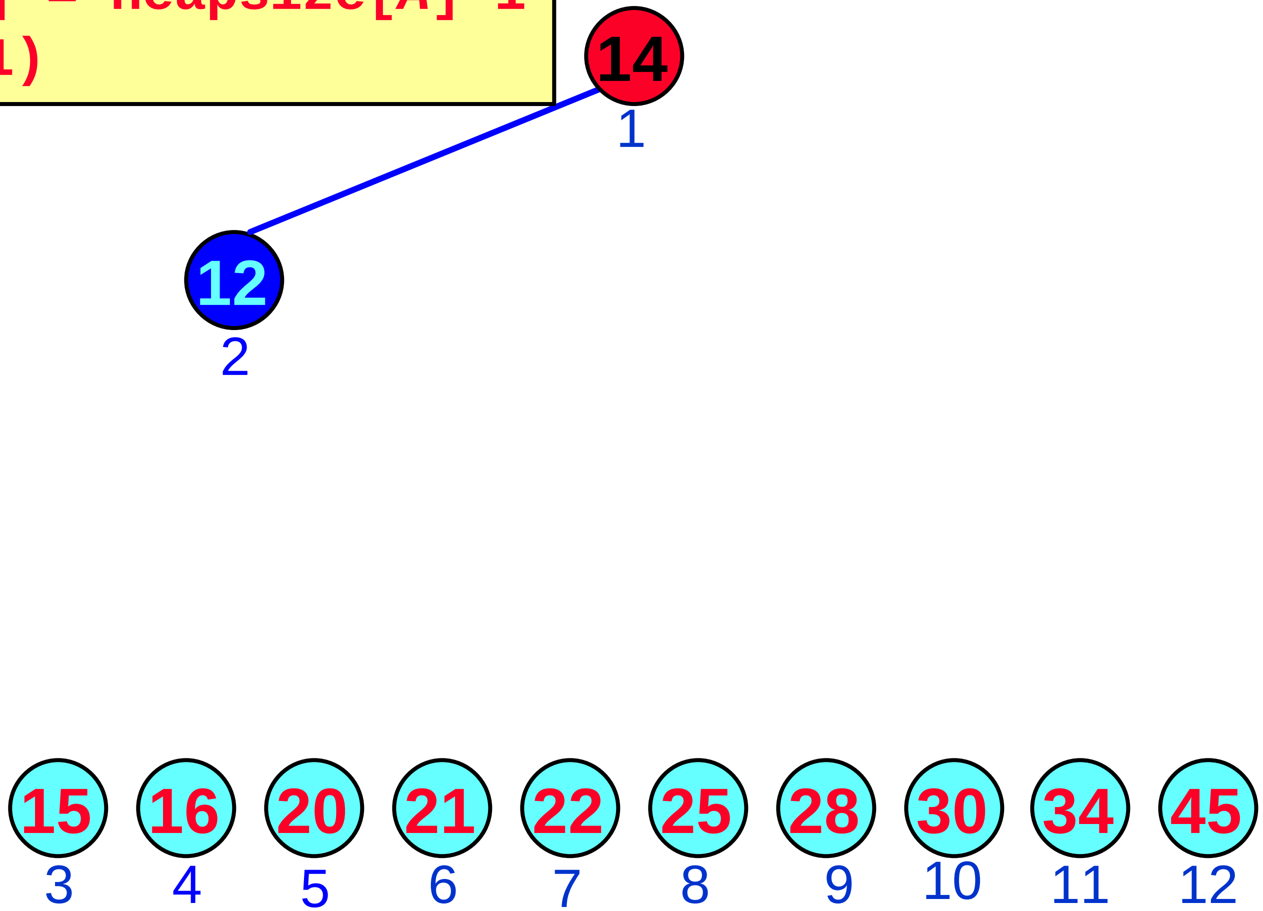


Heap Sort

```
Heap-Sort(A)
  …
  FOR i = length[A] DOWNTO 2
     DO "scambia A[1] e A[i]"
        heapsize[A] = heapsize[A]-1
        Heapify(A,1)
```

i = 3

heapsize[*A*]=2

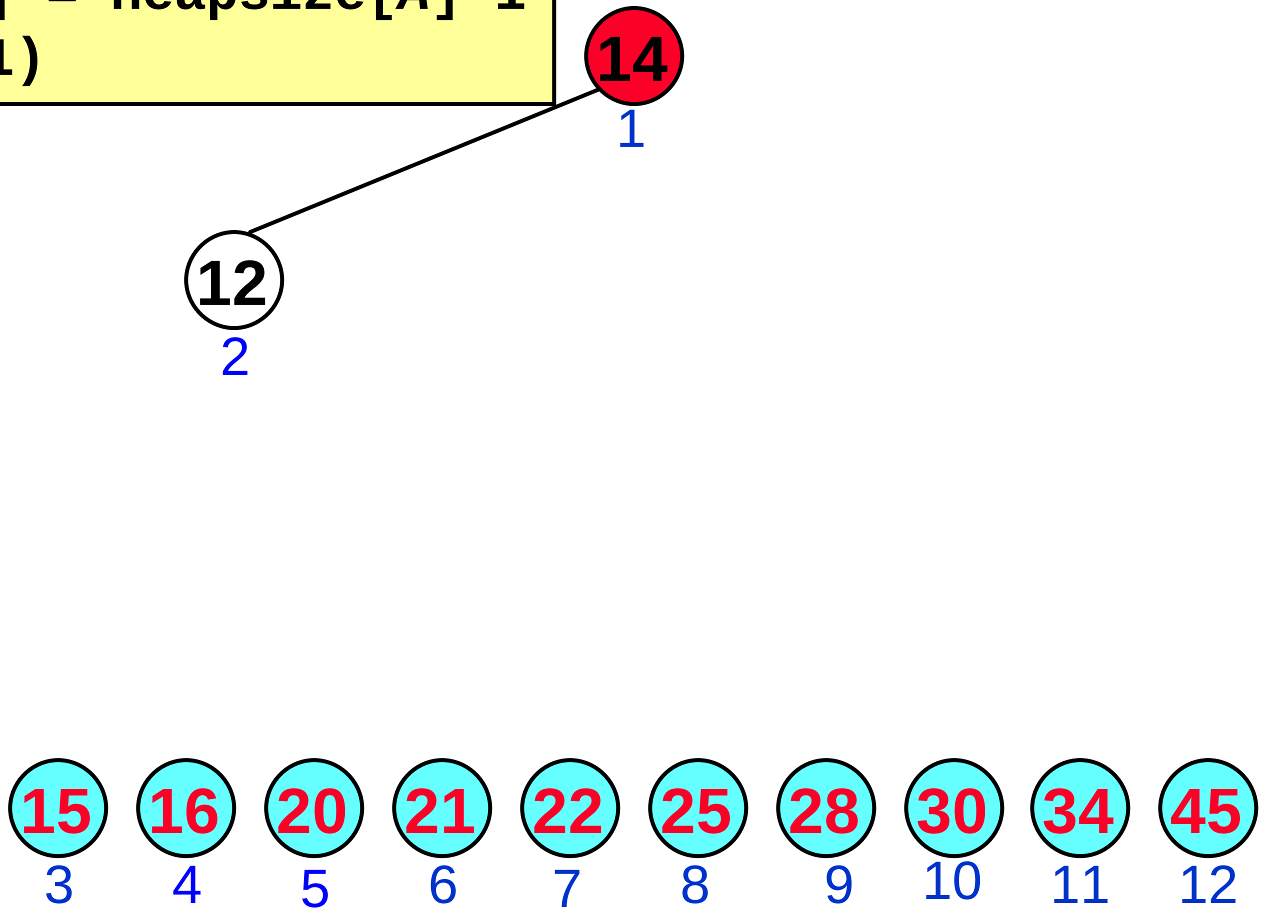


Heap Sort

```
Heap-Sort(A)
  …
  FOR i = length[A] DOWNTO 2
      DO "scambia A[1] e A[i]"
         heapsize[A] = heapsize[A]-1
         Heapify(A,1)
```

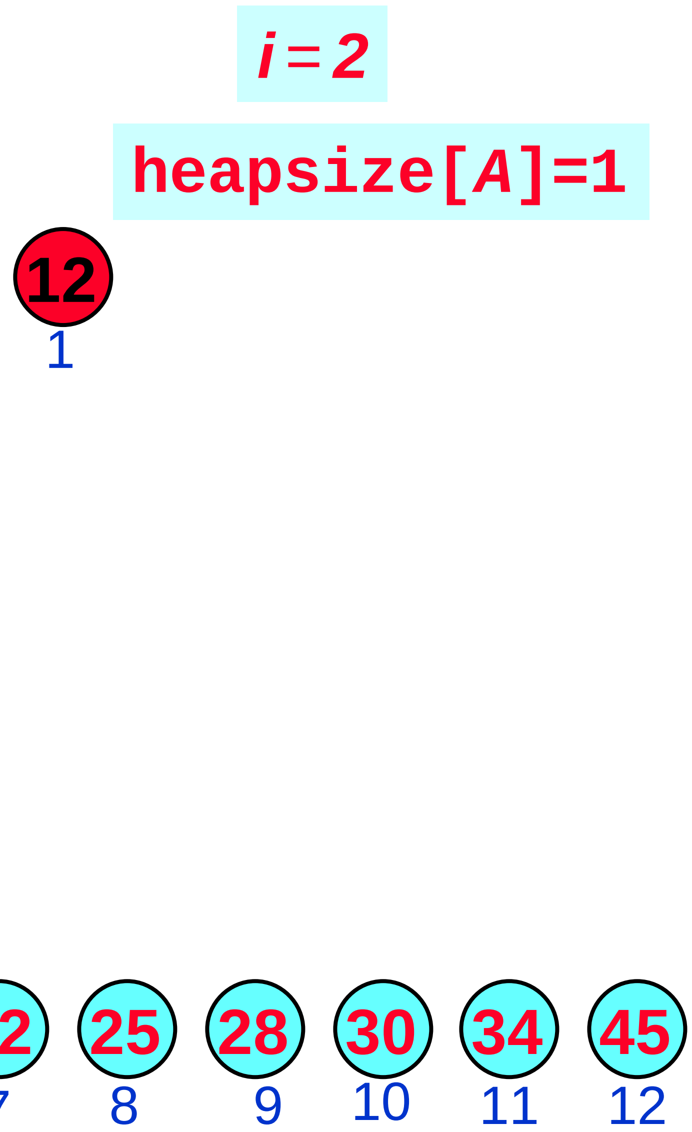
i = *2*

heapsize[*A*]=2



Heap Sort

```
Heap-Sort(A)
  …
  FOR i = length[A] DOWNTO 2
     DO "scambia A[1] e A[i]"
        heapsize[A] = heapsize[A]-1
        Heapify(A,1)
```



Heap Sort

```
Heap-Sort(A)
  …
  FOR i = length[A] DOWNTO 2
     DO "scambia A[1] e A[i]"
        heapsize[A] = heapsize[A]-1
        Heapify(A,1)
```

$i = 1$

heapsize[*A*]=1

12	14	15	16	20	21	22	25	28	30	34	45
1	2	3	4	5	6	7	8	9	10	11	12

Heap Sort

```
Heap-Sort(A)
  …
  FOR i = length[A] DOWNTO 2
     DO "scambia A[1] e A[i]"
        heapsize[A] = heapsize[A]-1
        Heapify(A,1)
```

$i = 1$

heapsize[*A*]=1

L'array *A* è ordinato!

1	2	3	4	5	6	7	8	9	10	11	12
12	14	15	16	20	21	22	25	28	30	34	45

Heap Sort

```
Heap-Sort(A)
   Costruisci-Heap(A)                } = O(n)
   FOR i = length[A] DOWNTO 2
      DO "scambia A[1] e A[i]"       }
         heapsize[A] = heapsize[A]-1 } = O(1)
         Heapify(A,1)                } = O(log n)
```

Complessità di Heap Sort

Nel caso peggiore *Heap-Sort* chiama

- ***una volta Costruisci-Heap*;**
- ***n-1* volte *Heapify* sullo *Heap* corrente**

$$T(n) = \max(O(n), (n-1) \times \max(O(1), T(\text{Heapify})))$$

Complessità di Heap Sort

Nel caso peggiore *Heap-Sort* chiama

- ***una volta Costruisci-Heap*;**
- ***n-1* volte *Heapify* sull'intero *Heap* .**

$$T(n) = max(O(n), (n-1) \times max(O(1), T(Heapify)))$$
$$= max(O(n), max(O(n), O(n \log n)))$$

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

HeapSort: conlusioni

HeapSort

- **Algoritmo di ordinamento *sul posto per confronto* che impiega tempo *$O(n \log n)$*.**

- **Algoritmo non immediato nè ovvio.**

- **Sfrutta le proprietà della struttura dati astratta *Heap*.**

HeapSort: conlusioni

HeapSort **dimostra che:**

- **scegliere una buona rappresentazione per i dati spesso facilita la progettazione di buoni algoritmi;**

- **importante pensare a quale può essere una buona rappresentazione dei dati prima di implementare una soluzione.**