Heapsort

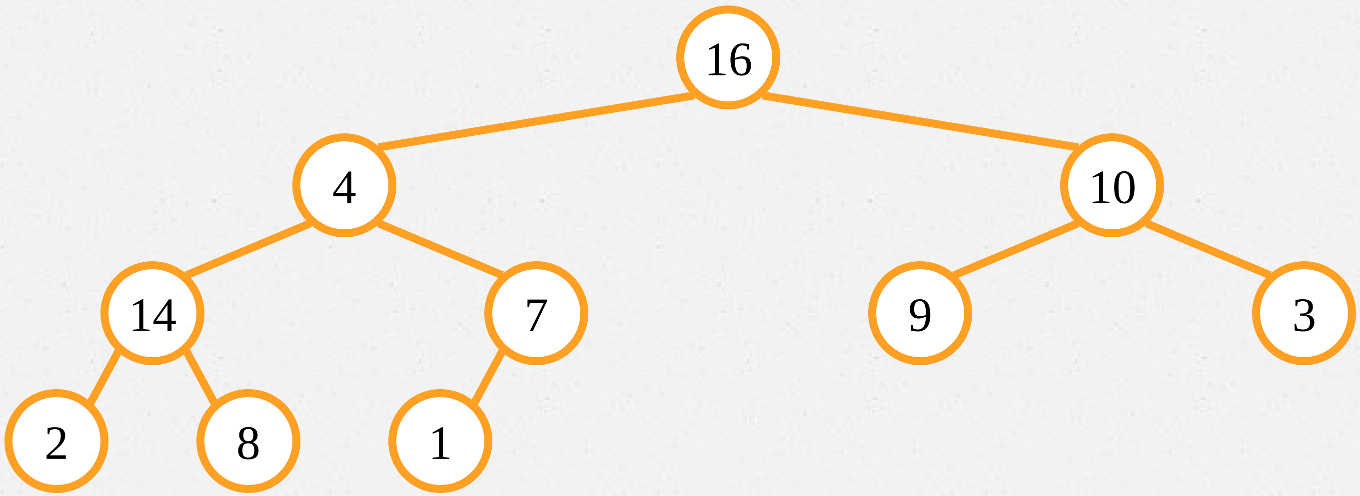
Leitura: Cormen – Capítulo 6

Heaps

- Uma estrutura de dados heap (binária) é um vetor de objetos que pode ser vista como uma árvore binária “aproximadamente” completa.

- A árvore é completamente preenchida em todos os níveis, exceto possivelmente pelo nível mais baixo que é preenchido a partir da esquerda até um certo ponto da estrutura.

Heaps



A =

16	4	10	14	7	9	3	2	8	1

Heaps

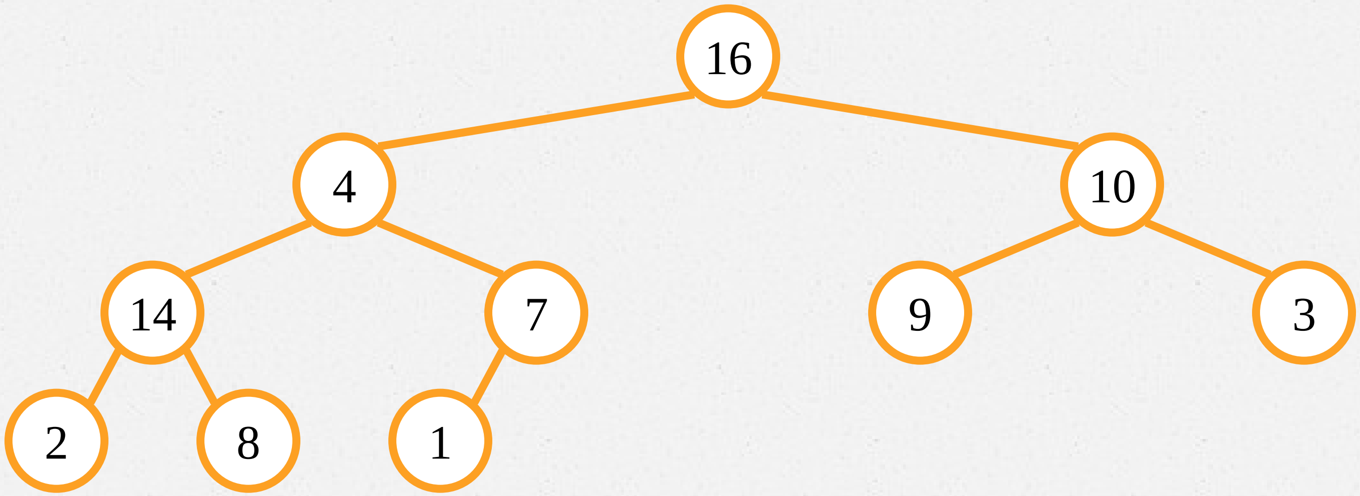
- Podemos representar uma heap como vetor, calculando os índices que ligam nós pais e filhos esquerdo e direito:

```
Parent(i)
1.return ⌊i/2⌋
```

```
Left(i)
1.return 2i
```

```
Right(i)
1.return 2i+1
```

Heaps



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

Propriedades da Heap

- **Max-heap** : *A[Parent(i)] ≥ A[i]*
- **Min-heap** : *A[Parent(i)] ≤ A[i]*
- A altura de um nó na árvore é dada pelo número de arestas no caminho mais longo do nó atual até um nó folha.
- A altura da árvore é dada pela altura do nó raiz.
- A altura de uma heap de n elementos, baseada em uma árvore binária completa, será $\Theta(\lg n)$.

Rotinas para heap

- Max-Heapify
- Build-Max-Heap
- Heapsort
- Max-Heap-Insert
- Heap-Extract-Max
- Heap-Increase-Key
- Heap-Maximum

Max-Heapify(A, i)

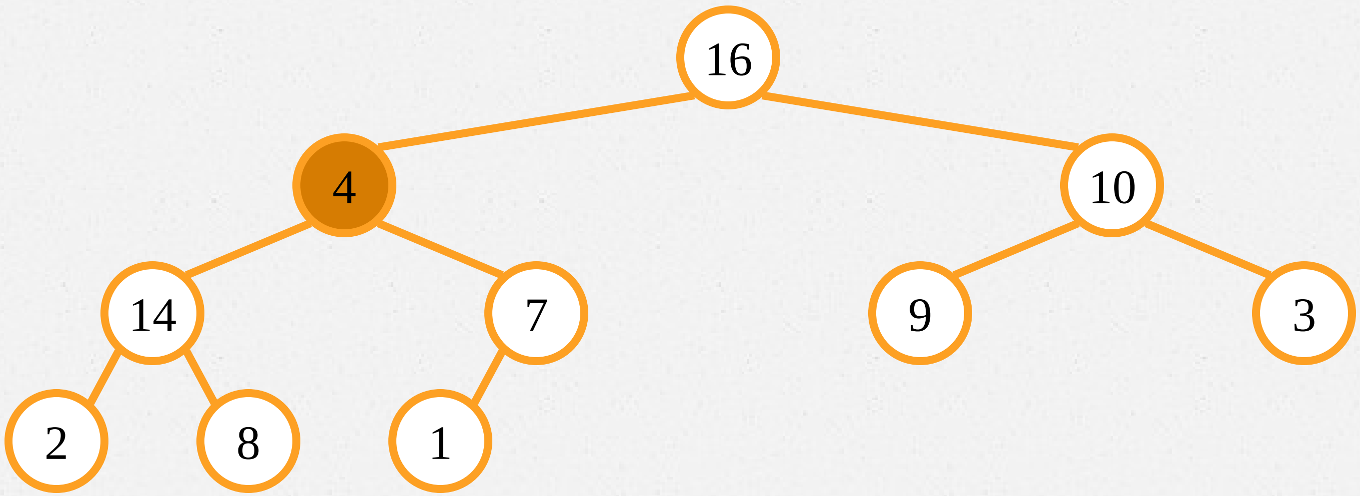
- Essa rotina assegura a propriedade max-heap.

$$T(n) \le T\left(\frac{2n}{3}\right) + \Theta(1) \Rightarrow T(n) = O(\lg n)$$

```
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] ↔ A[largest]
          Max-Heapify(A, largest
```

Max-Heapify(A,i)

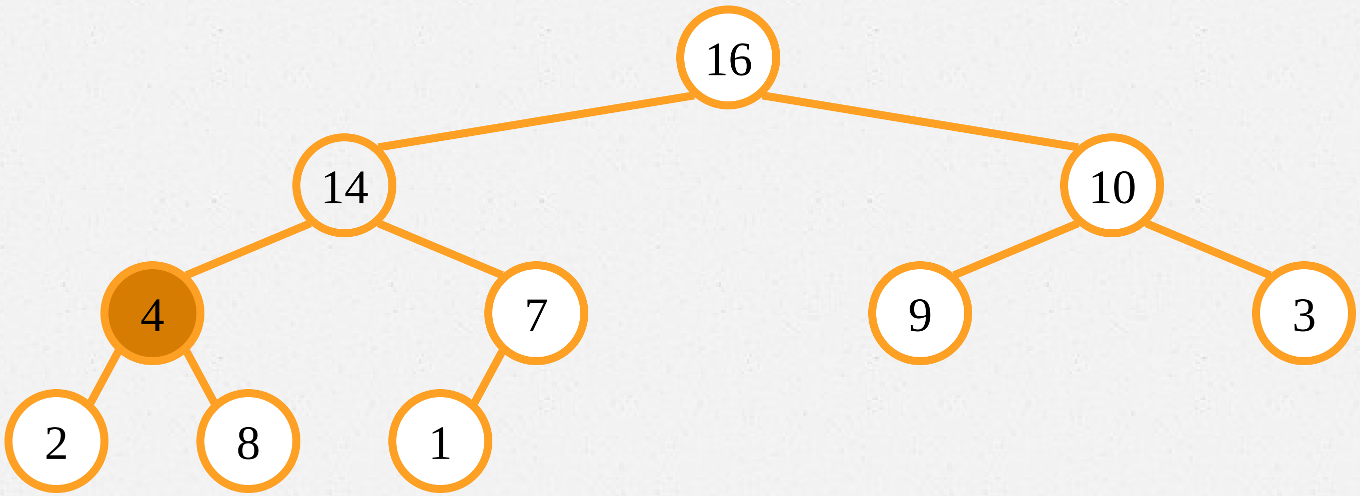
Max-Heapify(A,2)



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

Max-Heapify(A,i)

```
Max-Heapify(A,2)
   -> Max-Heapify(A,4)
```



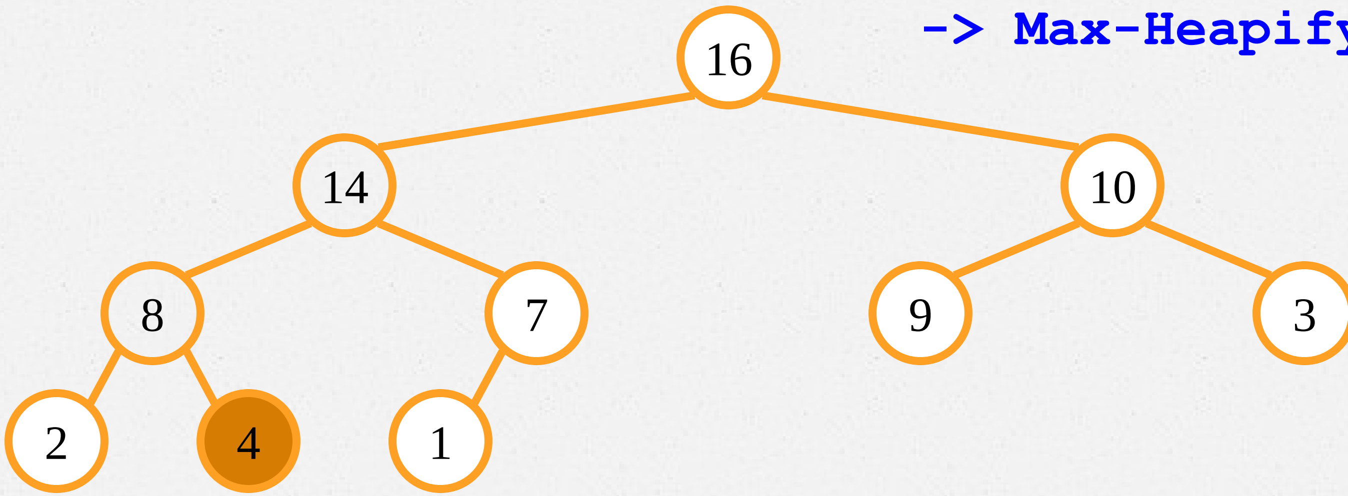
	1	2	3	4	5	6	7	8	9	10
A =	16	14	10	4	7	9	3	2	8	1

Max-Heapify(A,i)

`Max-Heapify(A,2)`

`-> Max-Heapify(A,4)`

`-> Max-Heapify(A,9)`



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

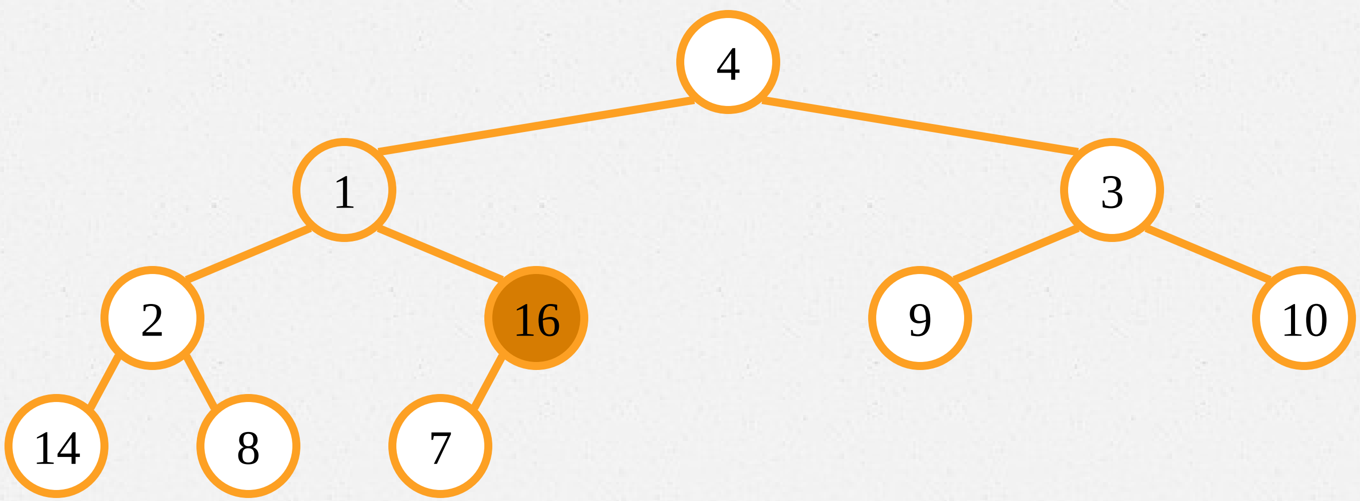
Build-Max-Heap(A)

- Retorna uma max-heap a partir de um vetor desordenado.

```
Build-Max-Heap(A)
A.heap-size = A.length
   for i = ⌊A.length/2⌋ downto 1
        Max-Heapify(A, i)
```

Build-Max-Heap(A)

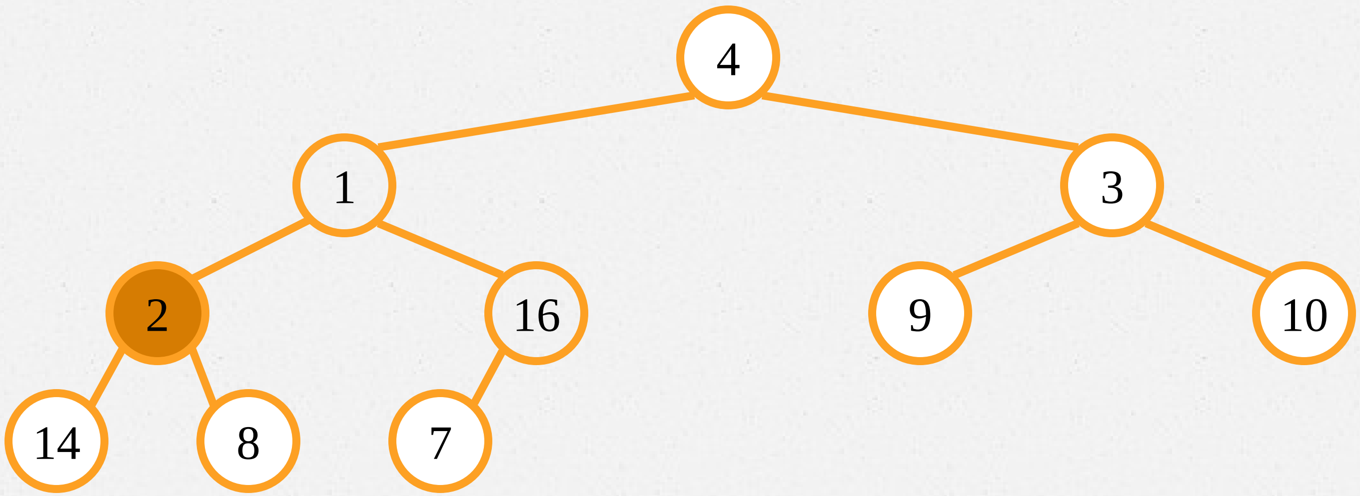
Max-Heapify(A,5)



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

Build-Max-Heap(A)

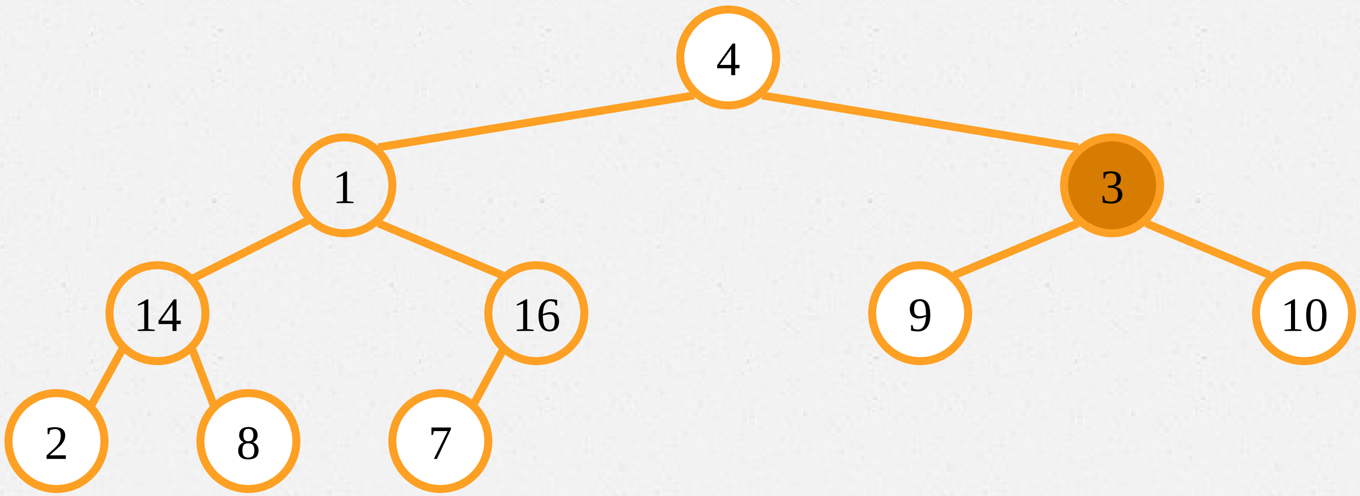
`Max-Heapify(A,4)`



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

Build-Max-Heap(A)

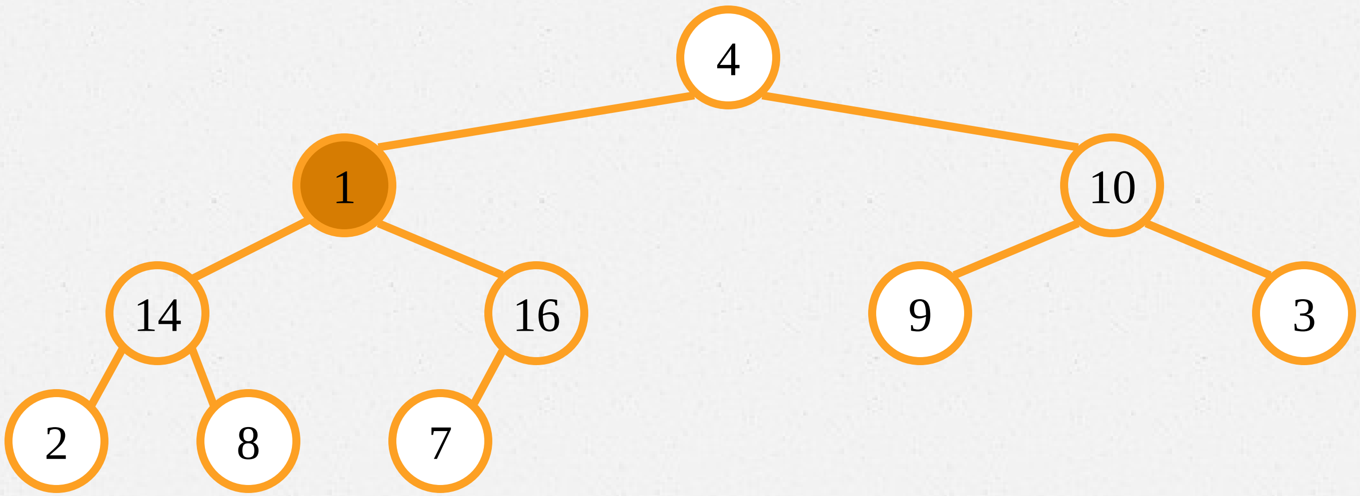
`Max-Heapify(A,3)`



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

Build-Max-Heap(A)

`Max-Heapify(A,2)`

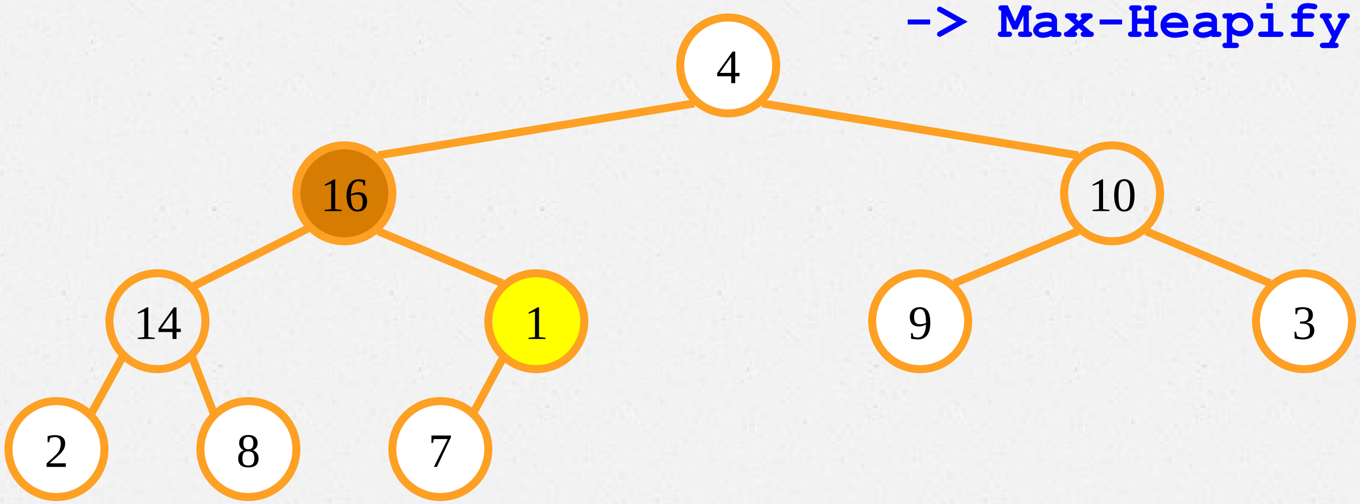


	1	2	3	4	5	6	7	8	9	10
A =	4	1	10	14	16	9	3	2	8	7

Build-Max-Heap(A)

Max-Heapify(A,2)

-> Max-Heapify(A,5)



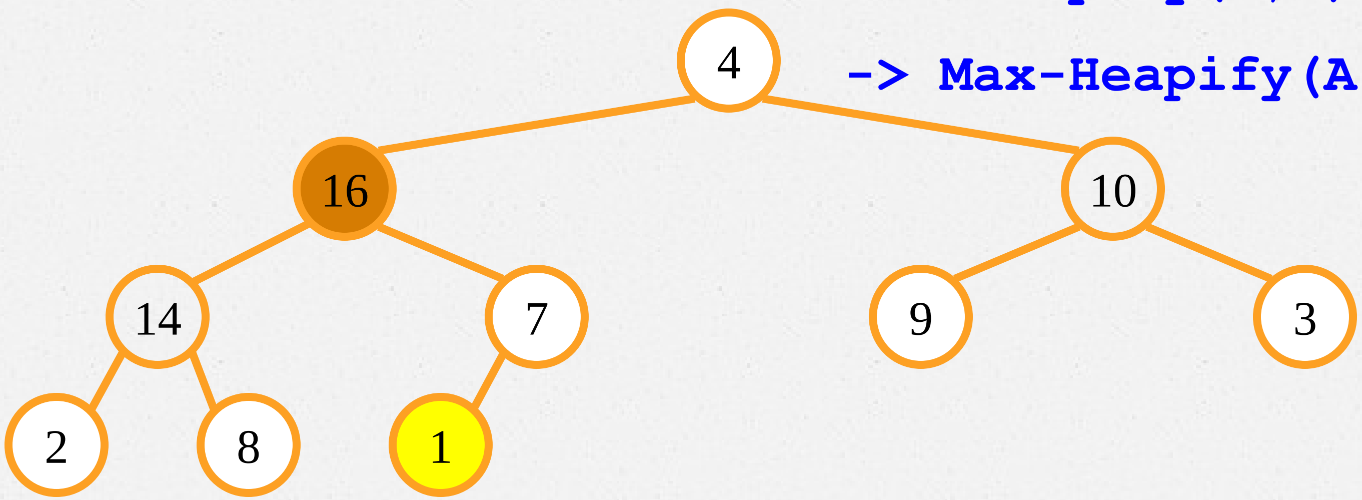
	1	2	3	4	5	6	7	8	9	10
A =	4	16	10	14	1	9	3	2	8	7

Build-Max-Heap(A)

Max-Heapify(A,2)

-> Max-Heapify(A,5)

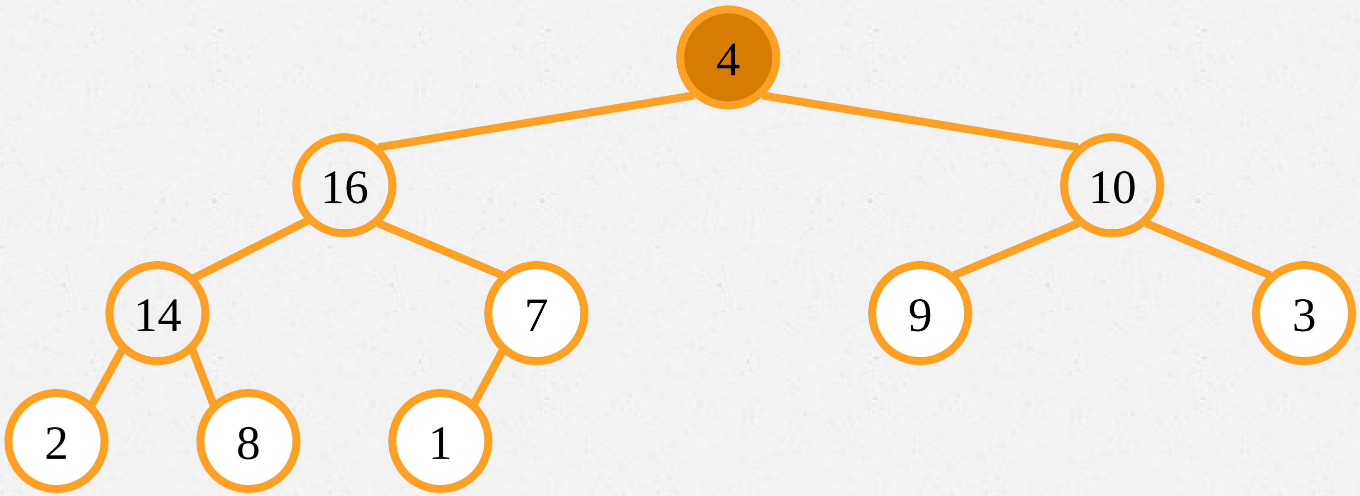
-> Max-Heapify(A,10)



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

Build-Max-Heap(A)

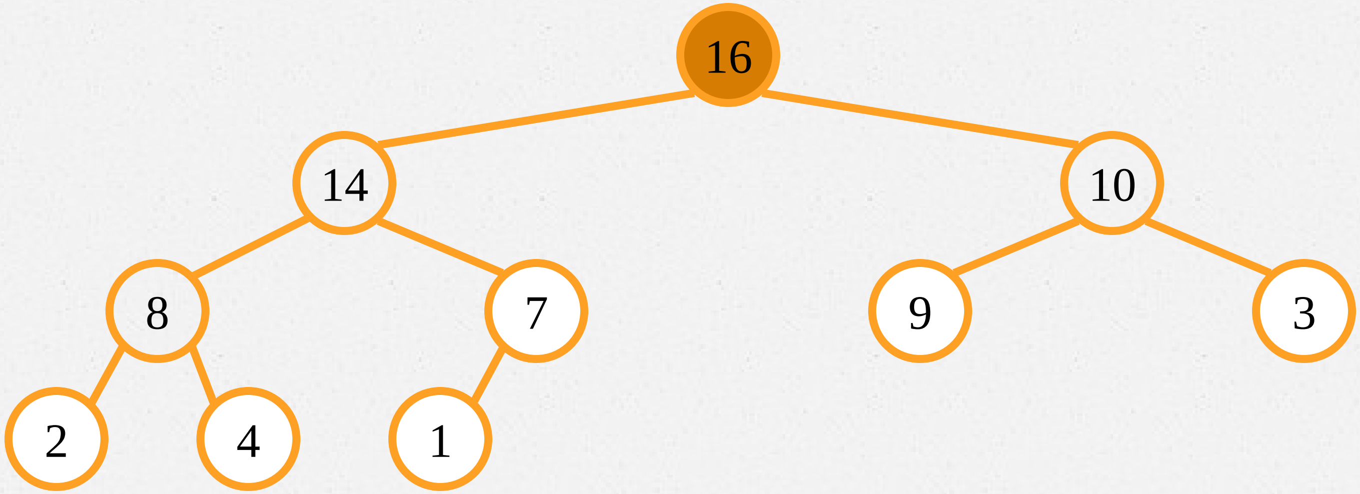
`Max-Heapify(A,1)`



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

Build-Max-Heap(A)

`Max-Heapify(A,1)`

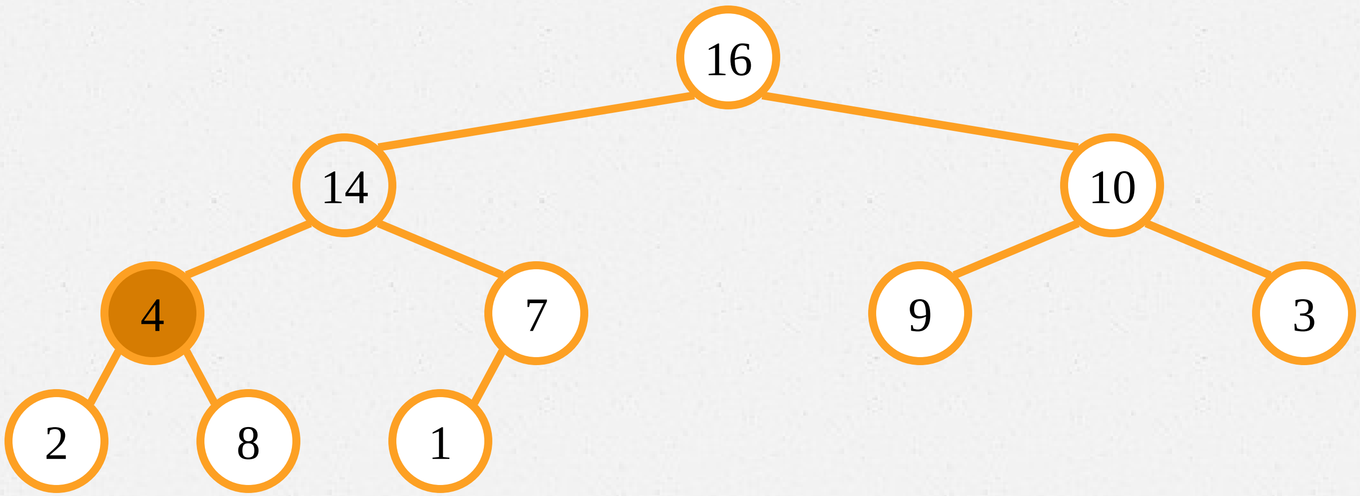


	1	2	3	4	5	6	7	8	9	10
A =	16	14	10	8	7	9	3	2	4	1

Build-Max-Heap(A)

Max-Heapify(A,2)

-> Max-Heapify(A,4)



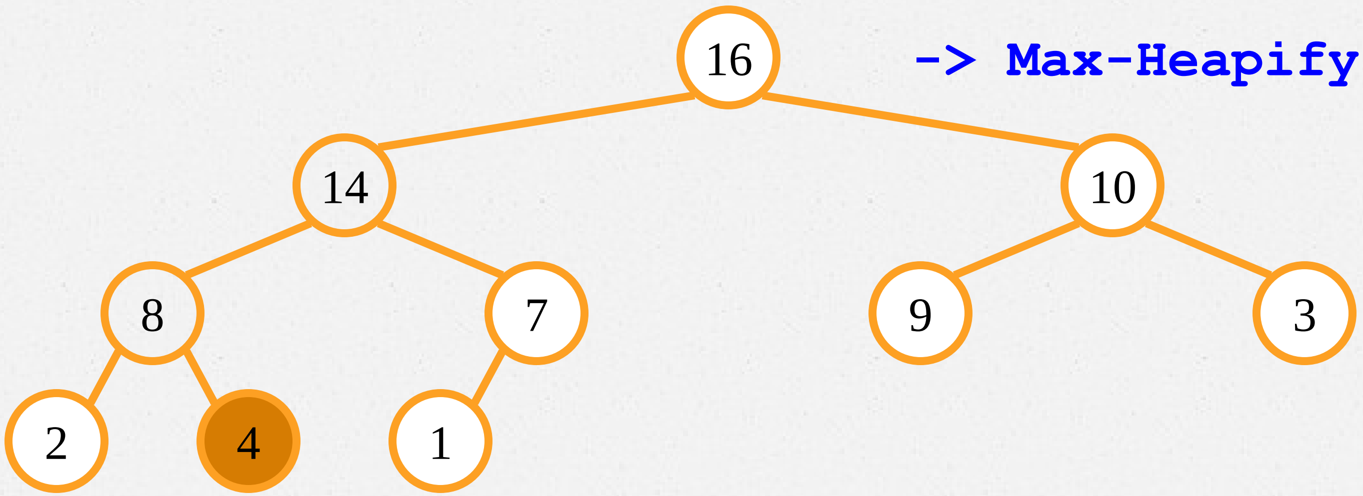
	1	2	3	4	5	6	7	8	9	10
A =	16	14	10	4	7	9	3	2	8	1

Build-Max-Heap(A)

`Max-Heapify(A,2)`

`-> Max-Heapify(A,4)`

`-> Max-Heapify(A,9)`



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

- $O(n \lg n)$?

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

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

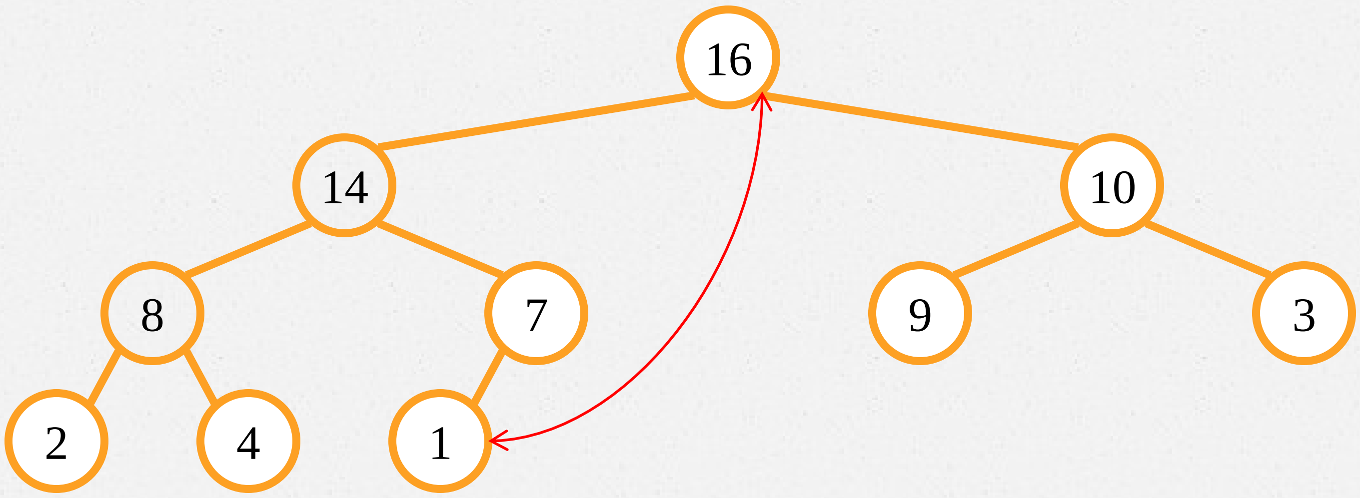
$$O(n \sum_{h=0}^{\lfloor \lg n \rfloor} \frac{h}{2^h}) = O(n \sum_{h=0}^{\infty} \frac{h}{2^h}) = O(n)$$

Build-Max-Heap não é O(nlgn), mas O(n)

Heapsort(A)

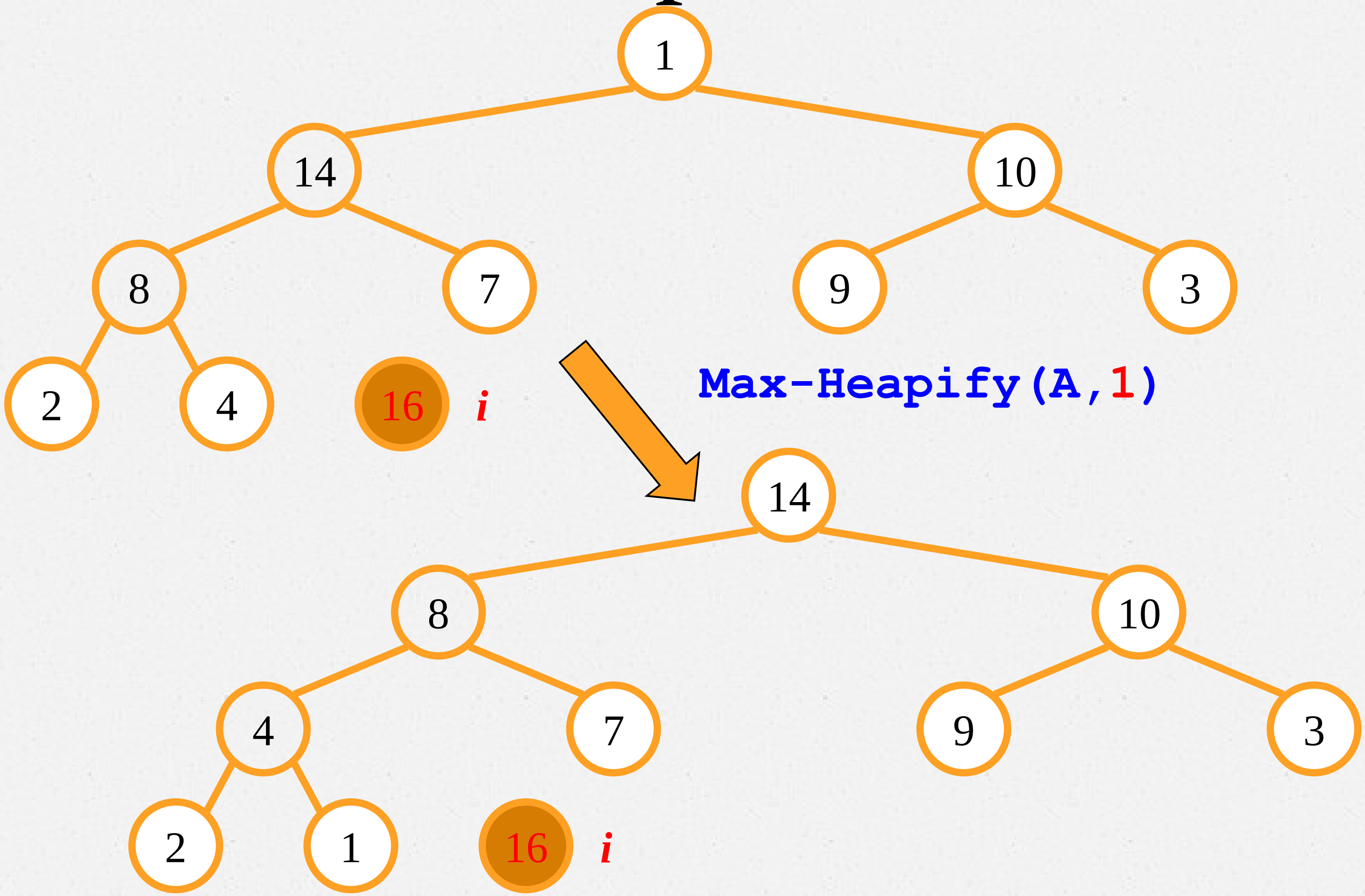
```
Heapsort(A)
Build-Max-Heap(A)
for i = A.length downto 2
    exchange A[1]↔A[i]
    A.heap-size = A.heap-size -1
    Max-Heapify(A,1)
```

Heapsort(A)

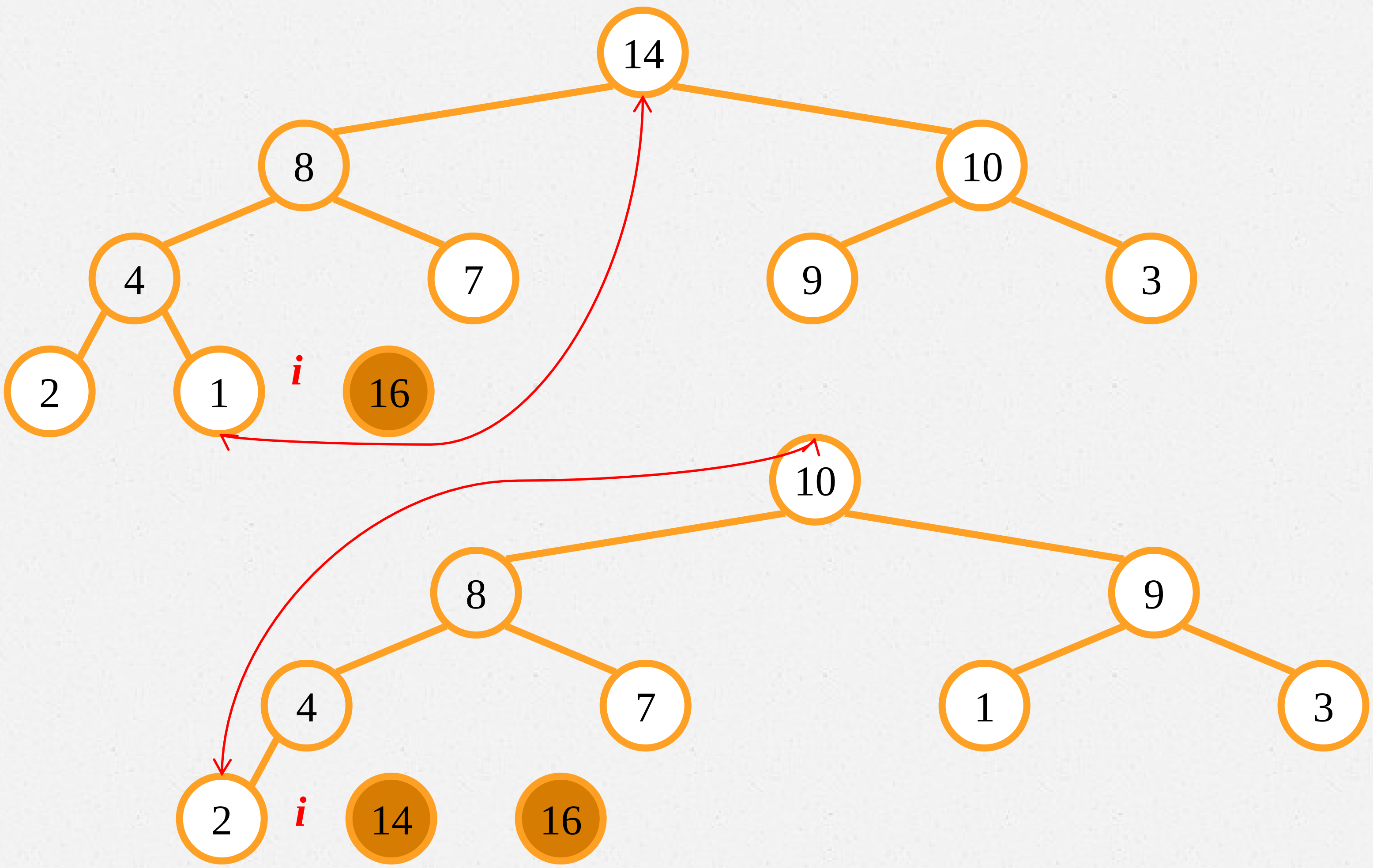


	1	2	3	4	5	6	7	8	9	10
A =	16	14	10	8	7	9	3	2	4	1

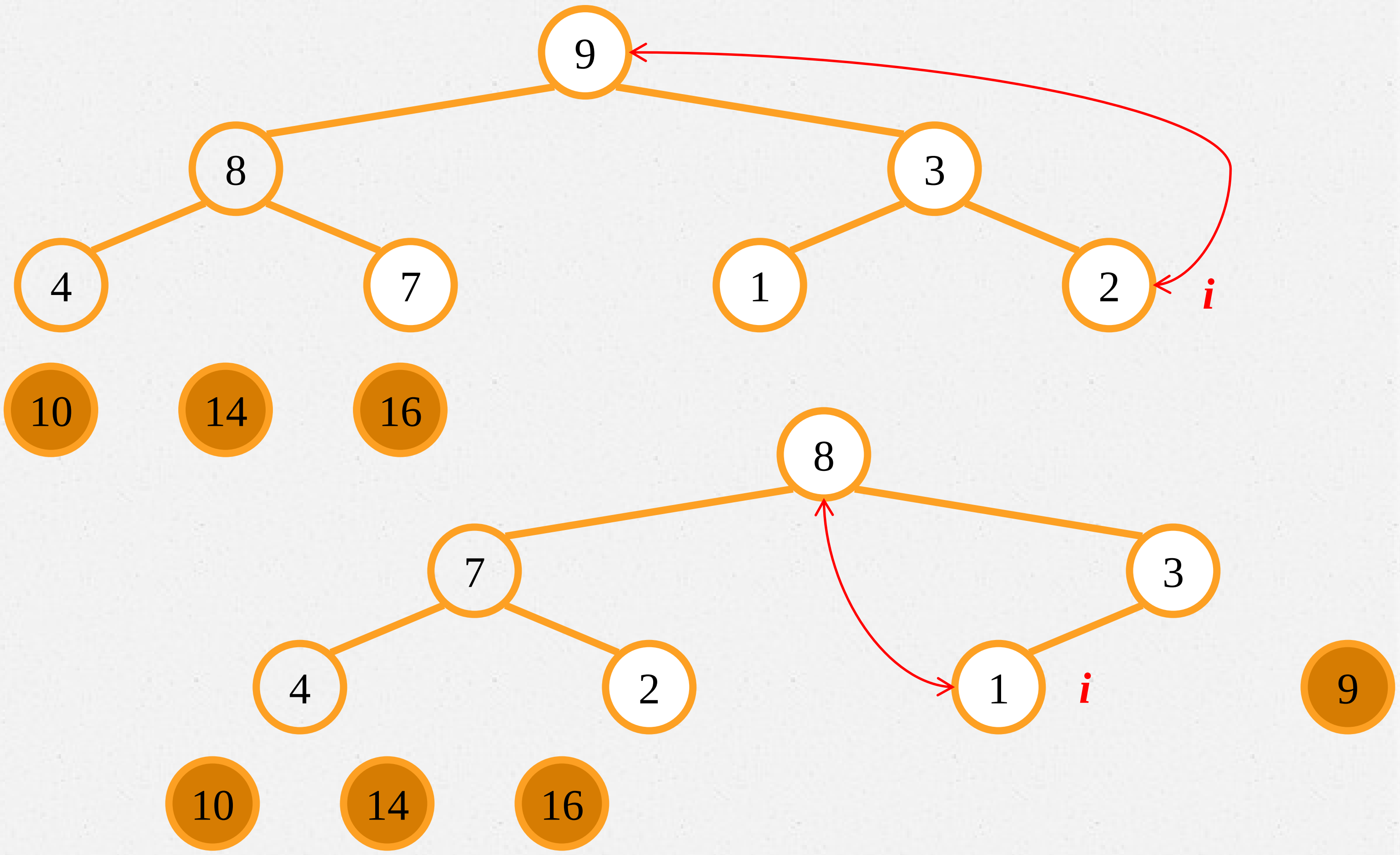
Heapsort(A)



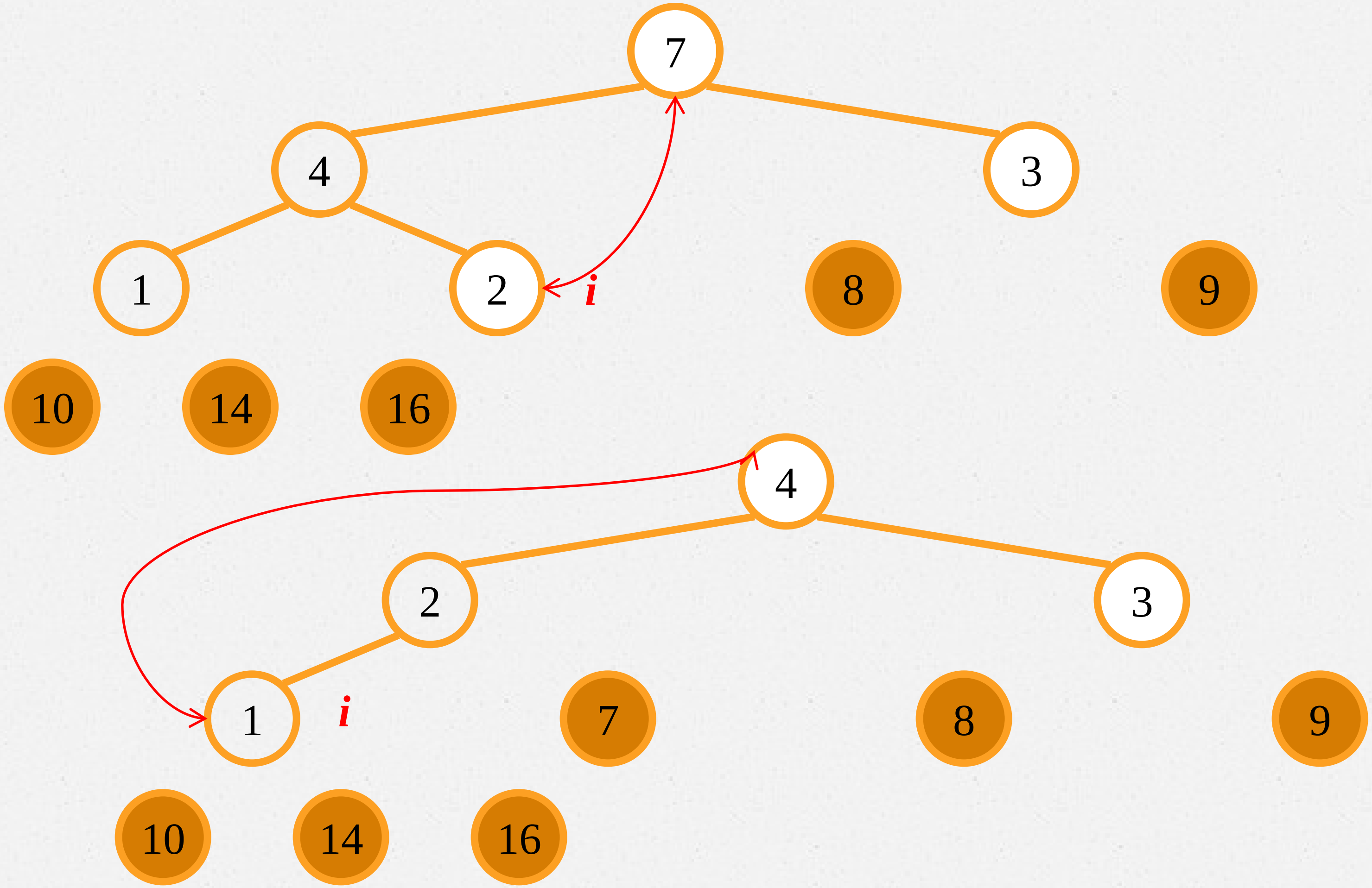
Heapsort(A)



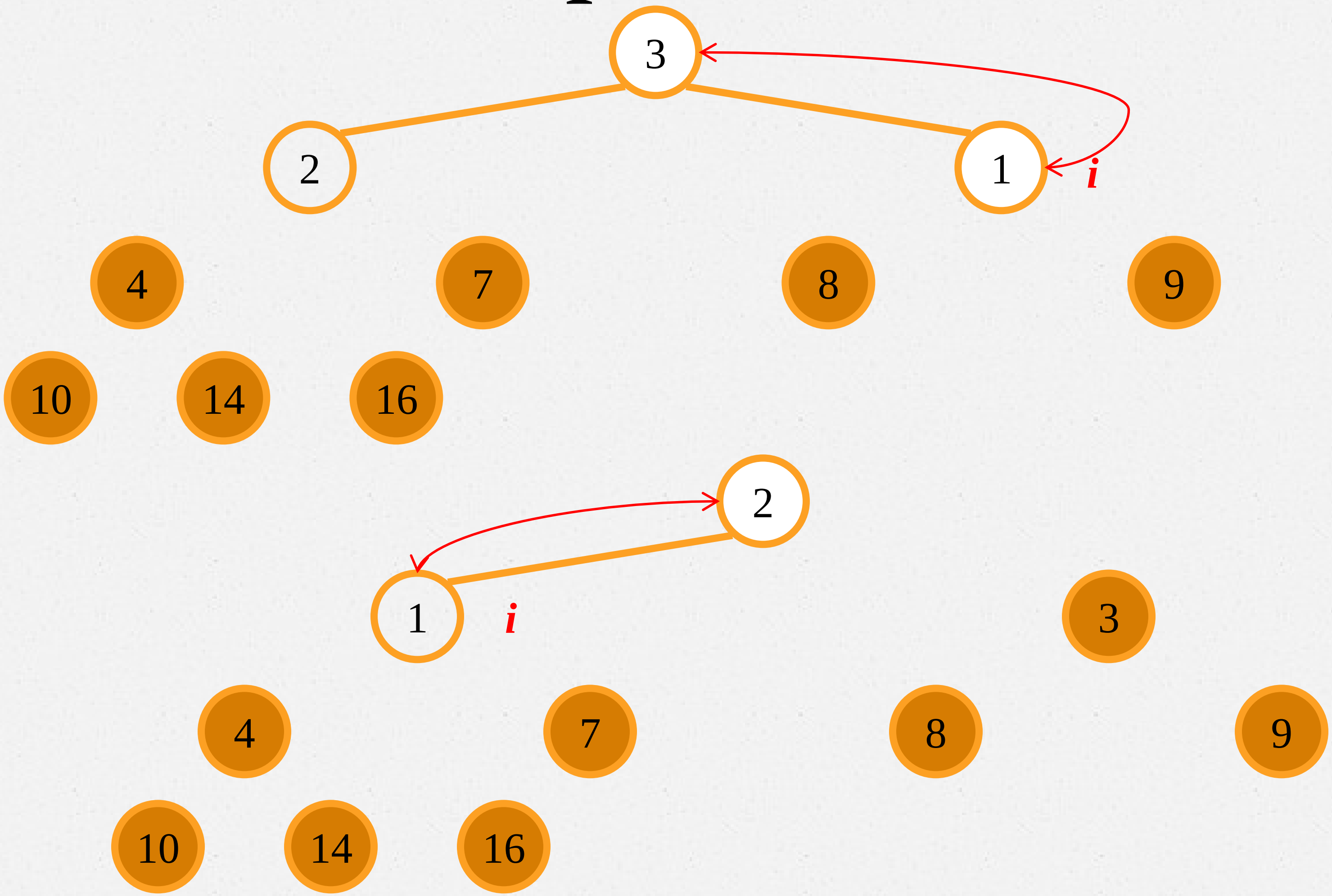
Heapsort(A)



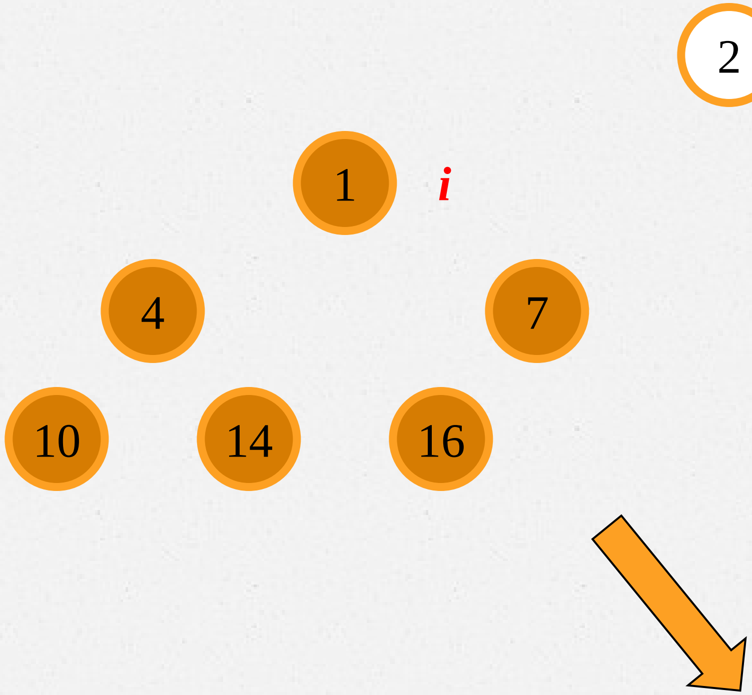
Heapsort(A)



Heapsort(A)



Heapsort(A)



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

Filas de Prioridade

- Filas de prioridade (priority queues) são estruturas de dados que mantem um conjunto *S* de elementos, onde cada um tem uma chave associada.
- A chamada **max-priority queue** suporta as seguintes operações:
  - **Insert(*S*, *x*)**
    - Insere o elemento *x* no conjunto *S*.
    - Complexidade O(lg *n*)
  - **Maximum(*S*)**
    - Retorna o elemento de *S* com a maior chave.
    - Complexidade O(1)

Filas de Prioridade

- **Extract-Max(S)**
  - **Remove e retorna o elemento de S com a maior chave.**
  - **Complexidade O(lg n)**

- **Increase-Key(S,x,k)**
  - **Aumenta o valor da chave de um elemento x para um novo valor k, onde k $\geq$ valor atual da chave do elemento x.**
  - **Complexidade O(lg n)**

Filas de Prioridade

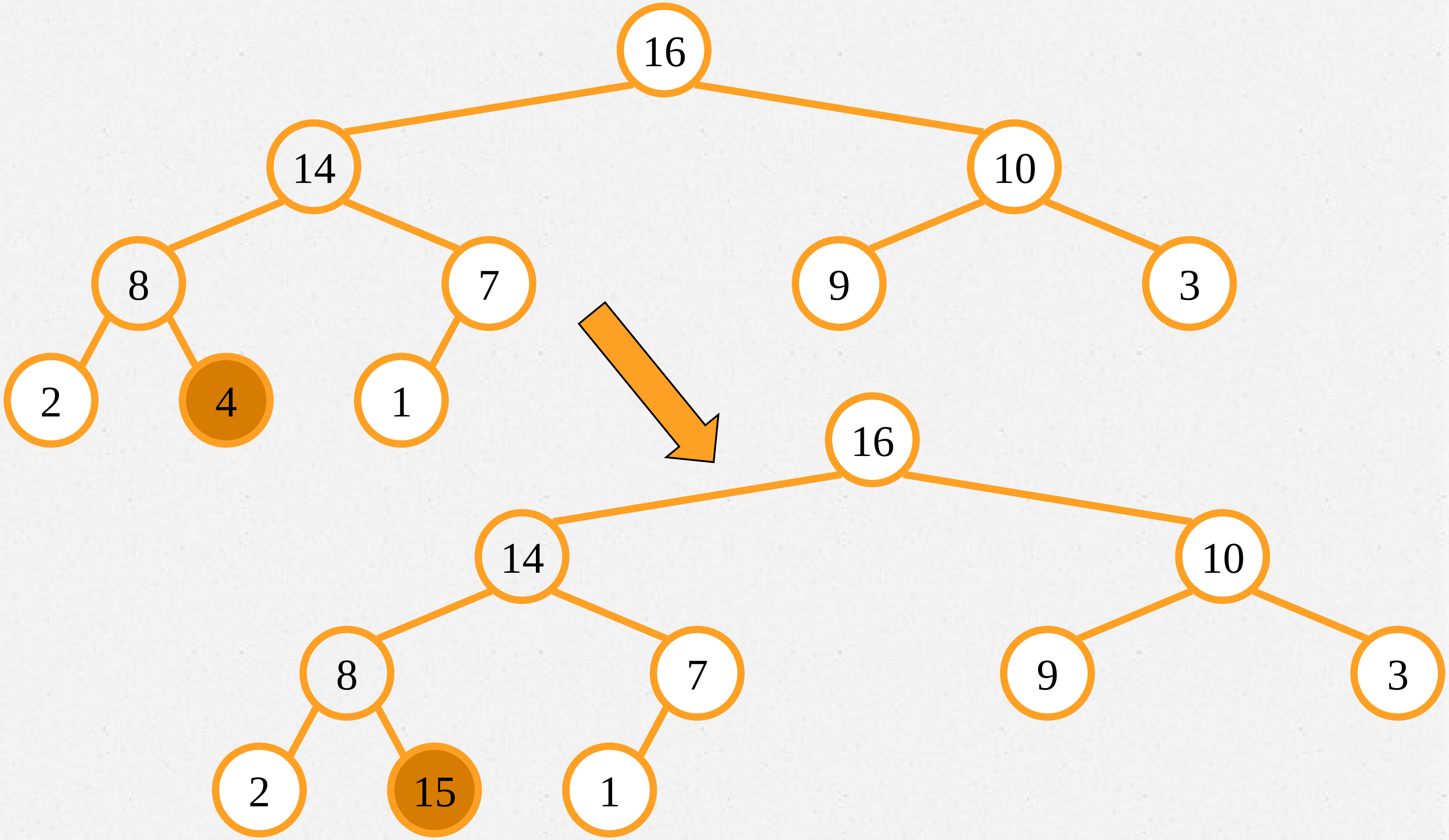
```
Heap-Maximum(A)
return A[1]
```

```
Heap_Extract-Max(A)
if heap-size[A] < 1
  error "heap underflow"
max = A[1]
A[1]=A[A.heap-size]
A.heap-size = A.heap-size- 1
Max-Heapify(A, 1)
return max
```

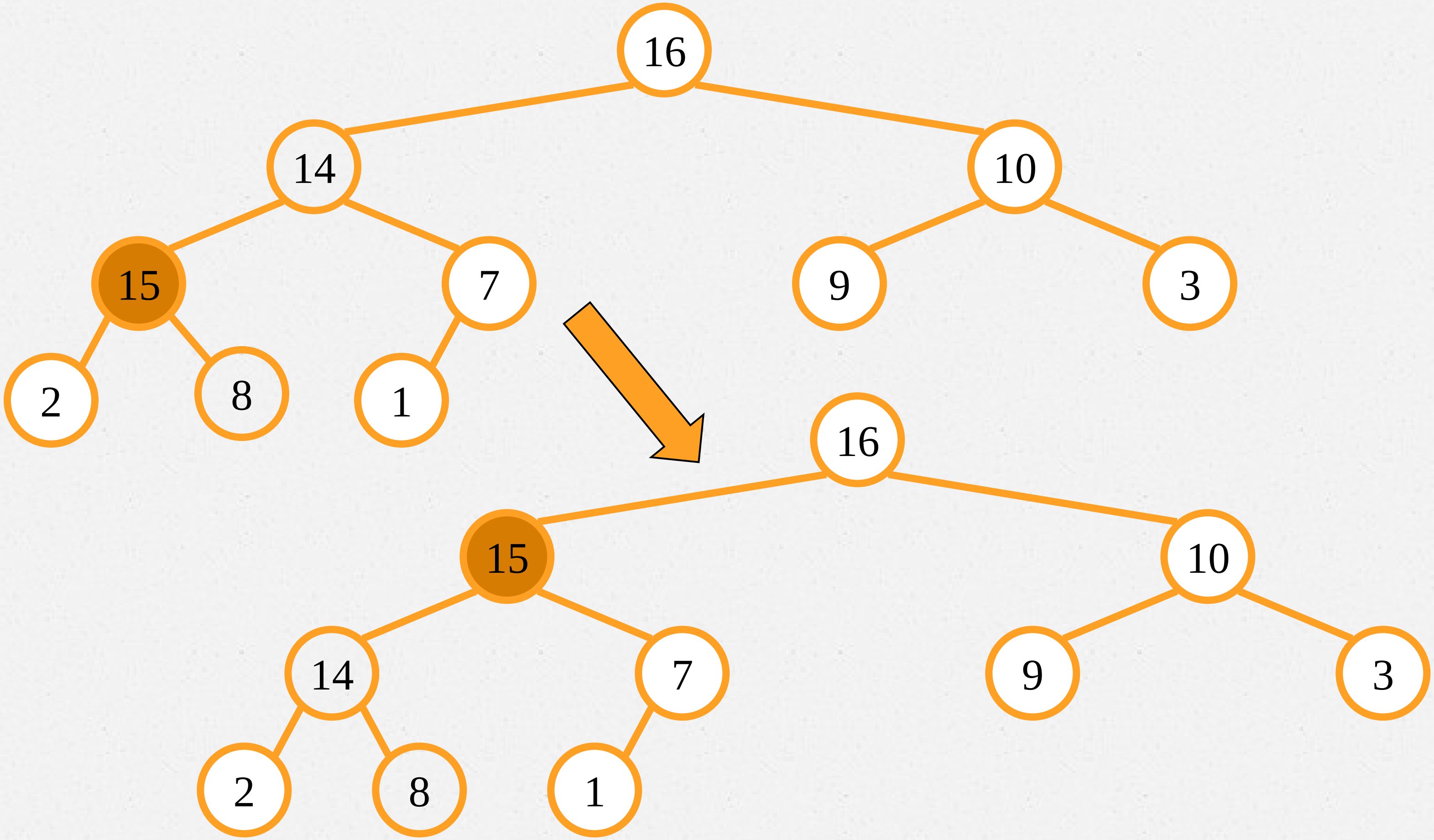
Heap-Increase-Key

```
Heap-Increase-Key (A, i, key)
   if key < A[i]
  error "new key is smaller than current key"
   A[i] = key
   while i > 1 and A[Parent(i)] < A[i]
         exchange A[i] ↔ A[Parent(i)]
         i = Parent(i)
```

Heap-Increase-Key



Heap-Increase-Key



Heap_Insert

```
Heap_Insert(A, key)
1  A.heap-size = A.heap-size + 1
2  A[A.heap-size] = -∞
3  Heap-Increase-Key(A, A.heap-size, key)
```