



# Métodos de Ordenação

---

Prof. Alneu de Andrade Lopes  
Prof. Thiago A. S. Pardo



# Ordenação por Seleção

---

- Idéia básica: os elementos são selecionados e dispostos em suas posições corretas
  - Seleção direta (ou simples), ou classificação de deslocamento descendente
  - Heap-sort, ou método do monte



# Seleção Direta

---

- Método

1. Selecionar o elemento que apresenta o menor valor
2. Trocar o elemento de lugar com o primeiro elemento da seqüência,  $x[0]$
3. Repetir as operações 1 e 2, envolvendo agora apenas os  $n-1$  elementos restantes, depois os  $n-2$  elementos, etc., até restar somente um elemento, o maior deles



# Seleção Direta

---

■  $x = 44, 55, 12, 42, 94, 18, 06, 67$

■ (vetor original)	44	55	12	42	94	18	06	67
■ passo 1 (06)	06	55	12	42	94	18	44	67
■ passo 2 (12)	06	12	55	42	94	18	44	67
■ passo 3 (18)	06	12	18	42	94	55	44	67
■ passo 4 (42)	06	12	18	42	94	55	44	67
■ passo 5 (44)	06	12	18	42	44	55	94	67
■ passo 6 (55)	06	12	18	42	44	55	94	67
■ passo 7 (67)	06	12	18	42	44	55	67	94



# Seleção Direta

---

- No  $i$ -ésimo passo, o elemento com o menor valor entre  $x[i], \dots, x[n]$  é selecionado e trocado com  $x[i]$
- Como resultado, após  $i$  passos, os elementos  $x[1], \dots, x[i]$  estão ordenados



# Seleção Direta

---

- Exercício em grupos de 4 alunos  
(valendo nota)
  - Implementar a Seleção Direta
  - Calcular a complexidade



# Seleção Direta

---

```
void selecao(int x[], int n) {  
    int i, j, menor, index;  
    for (i = 0; i < n-1; i++) {  
        menor = x[ i ];  
        index = i;  
        for (j = i+1; j < n-1; j++) {  
            if (x[ j ] < menor) {  
                menor = x[ j ];  
                index = j;  
            }  
        }  
        x[ index ] = x[ i ];  
        x[ i ] = menor;  
    }  
}
```



# Seleção Direta

---

- No primeiro passo ocorrem  $n - 1$  comparações, no segundo passo  $n - 2$ , e assim por diante
  - Logo, no total, tem-se  $(n - 1) + (n - 2) + \dots + 1 = n * (n - 1) / 2$  comparações:  $O(n^2)$
- Não existe melhora se a entrada está completamente ordenada ou desordenada
- Exige pouco espaço
- É melhor que o Bubble-sort
- É útil apenas quando  $n$  é pequeno





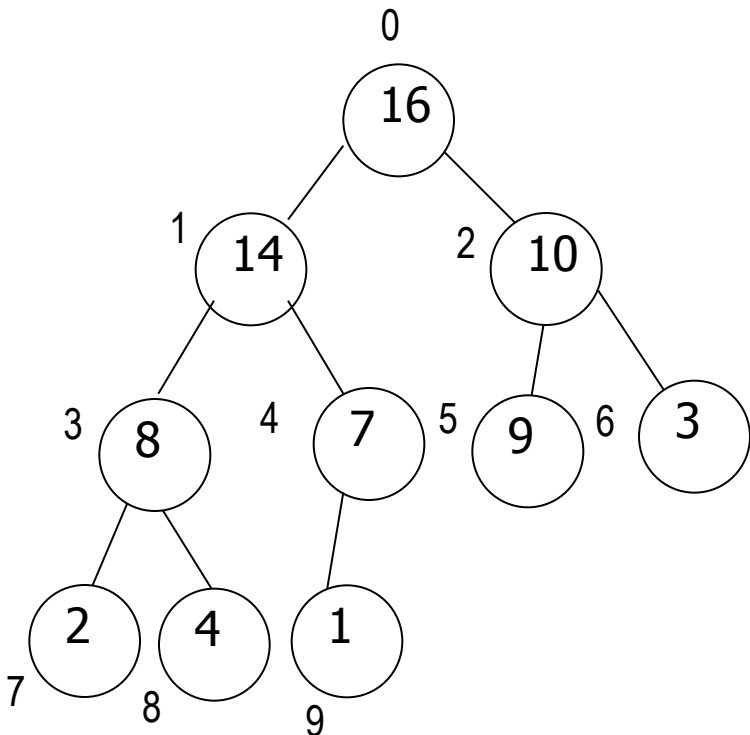
# Heap-sort

---

- Utiliza uma estrutura de dados - um heap – para ordenar os elementos
- Atenção: a palavra *heap* é utilizada atualmente em algumas linguagens de programação para se referir ao “espaço de armazenamento de lixo coletado”

# Heap-sort

- Um heap é um vetor que implementa (representa) uma árvore binária quase completa



0	1	2	3	4	5	6	7	8	9
16	14	10	8	7	9	3	2	4	1



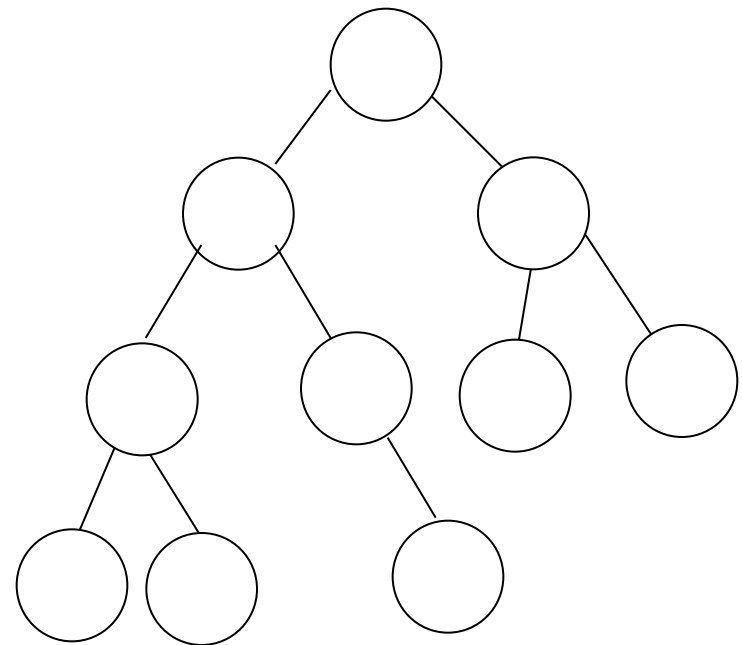
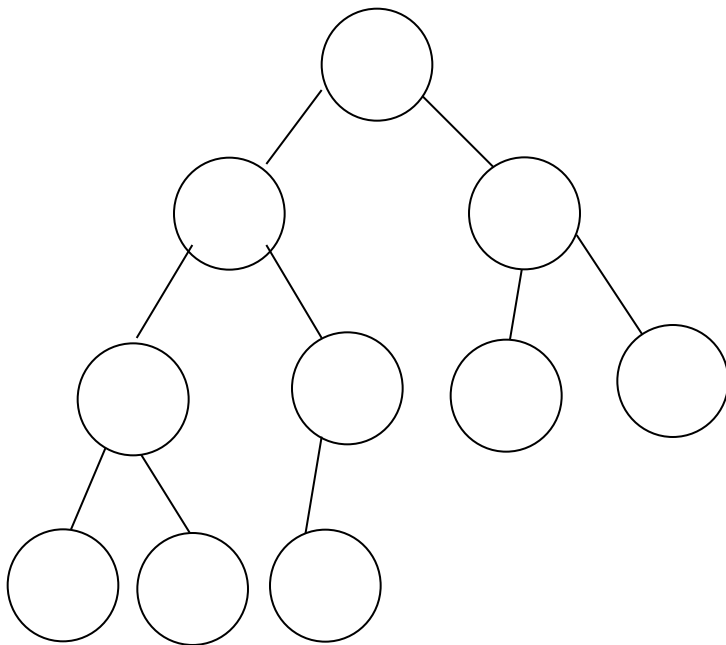
# Heap-sort

---

- Uma árvore binária de profundidade/nível  $d$  é uma árvore binária quase completa se:
  - Cada folha estiver no nível  $d$  ou  $d - 1$
  - Para cada nó  $v$  da árvore com descendente direito no nível  $d$ , todos os descendentes esquerdos de  $v$  que forem folhas estiverem também no nível  $d$

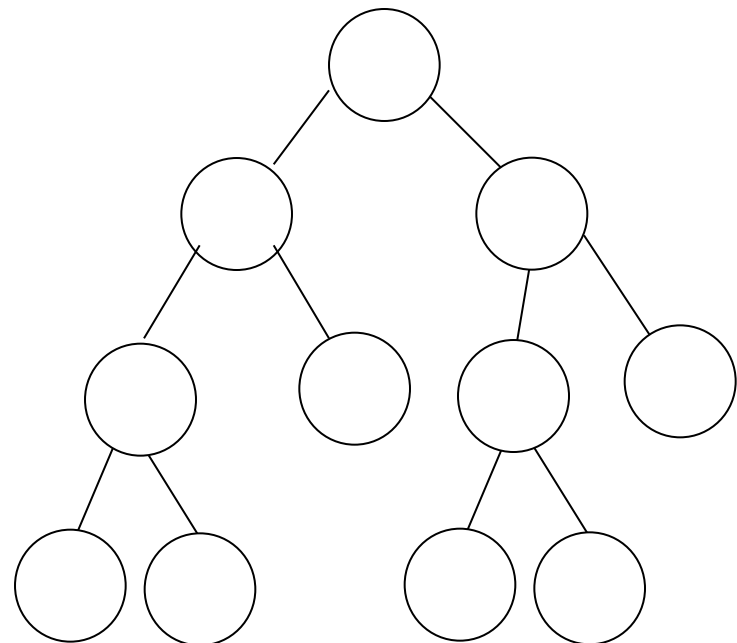
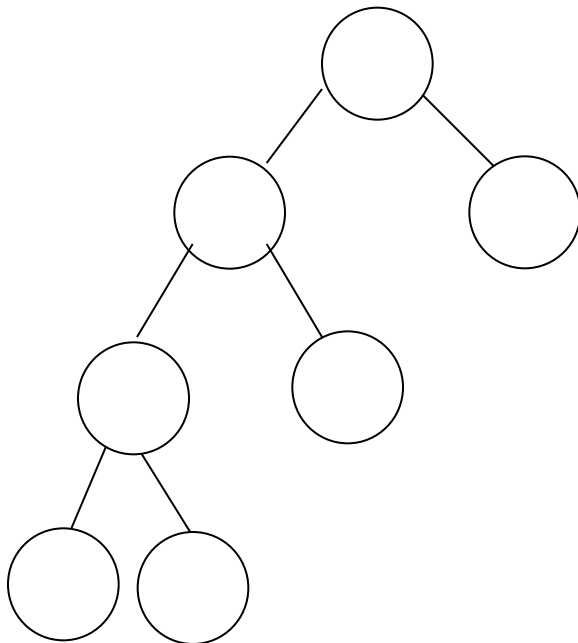
# Heap-sort

- Exemplos de árvores binárias quase completas



# Heap-sort

- Exemplos de árvores binárias que **não** são quase completas
  - Por quê?





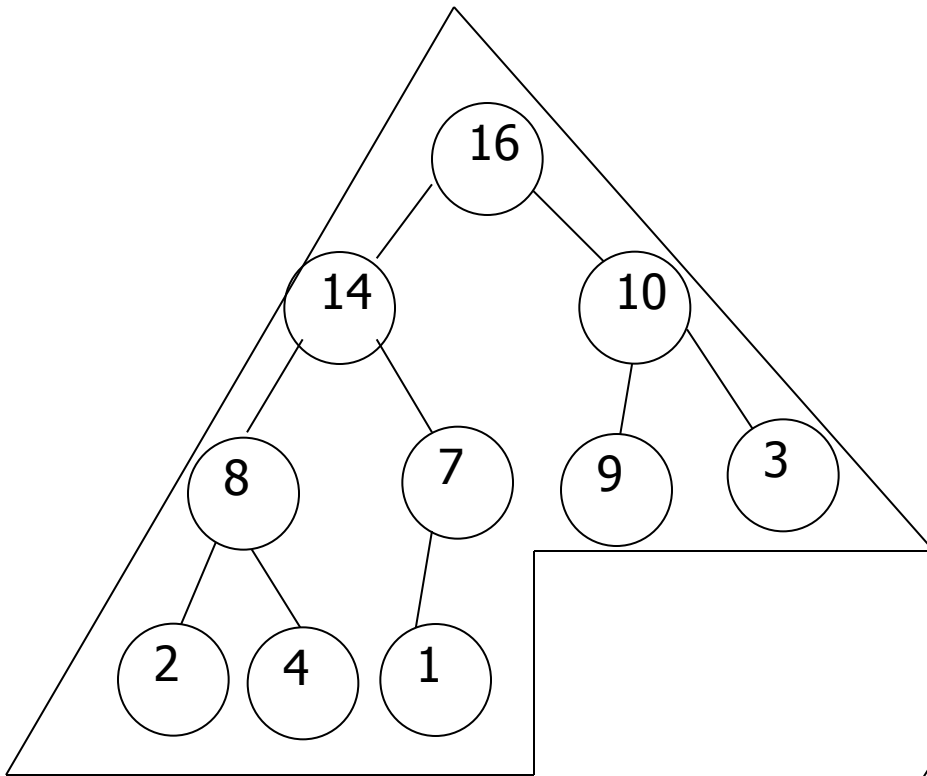
# Heap-sort

---

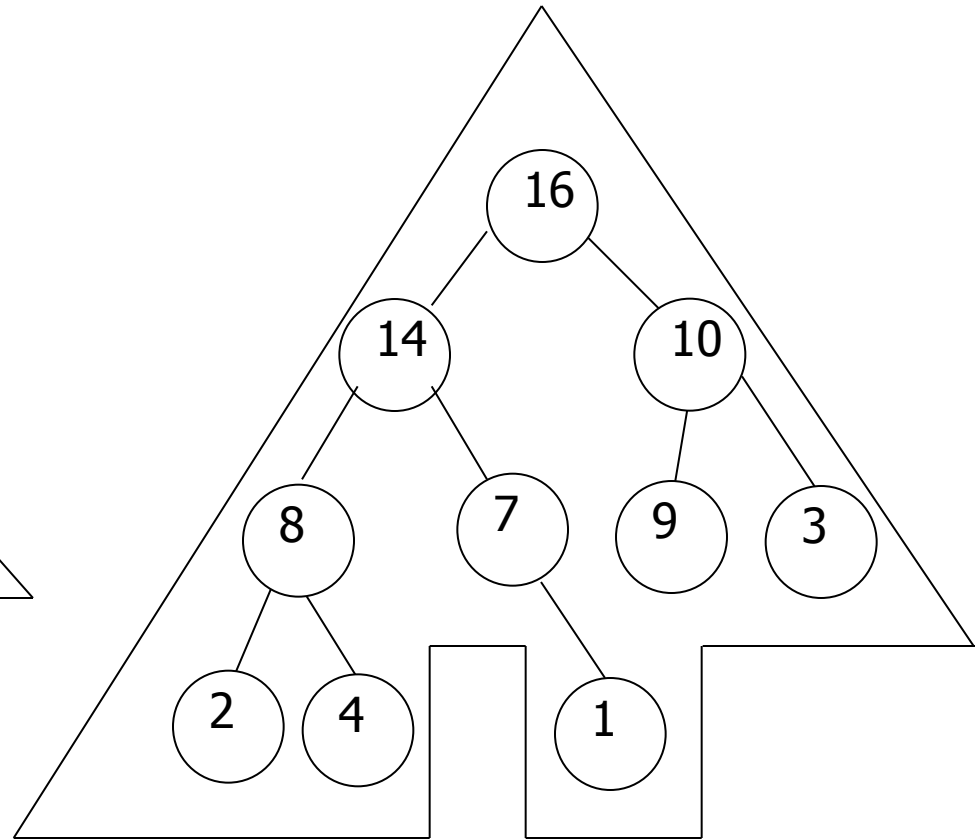
- Um heap observa conceitos de ordem e de forma
  - Ordem: o item de qualquer nó deve satisfazer uma relação de ordem com os itens dos nós filhos
    - **Heap máximo** (ou descendente):  $\text{pai} \geq \text{filhos}$ , sendo que a raiz é o maior elemento
      - *Propriedade de heap máximo*
    - Heap mínimo (ou heap ascendente):  $\text{pai} \leq \text{filhos}$ , sendo que a raiz é o menor elemento
      - *Propriedade de heap mínimo*
  - Forma: a árvore binária tem seus nós-folha, no máximo em dois níveis, sendo que as folhas devem estar o mais à esquerda possível



# Heap-sort



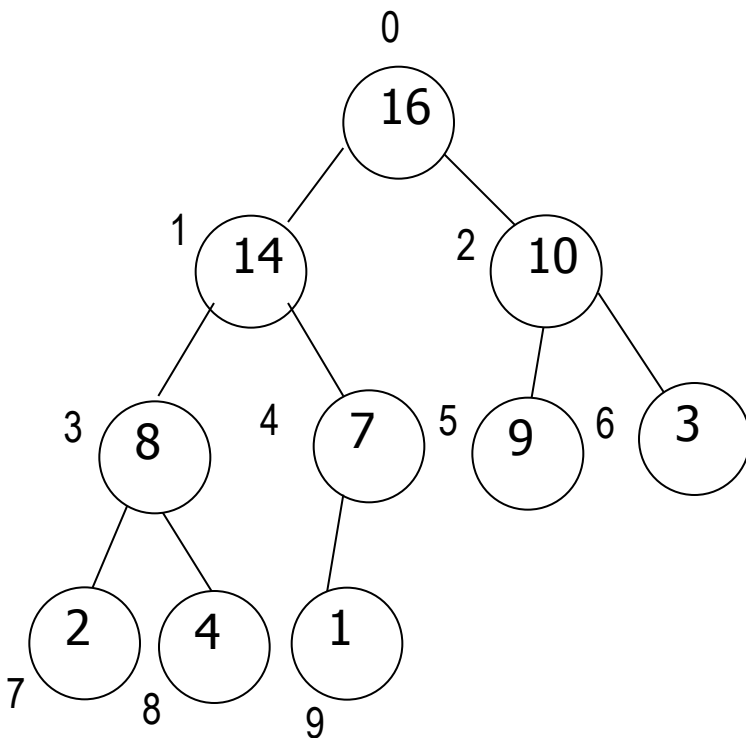
É um heap máximo



Não é um heap máximo<sub>15</sub>

# Heap-sort

- Como acessar os elementos (pai e filhos de cada nó) no heap?



0	1	2	3	4	5	6	7	8	9
16	14	10	8	7	9	3	2	4	1

Filhos do nó  $k$ :

- filho esquerdo =  $2k + 1$
- filho direito =  $2k + 2$

Pai do nó  $k$ :  $(k-1)/2$

Folhas de  $n/2$  em diante





# Heap-sort

---

- Assume-se que:
  - A raiz está sempre na posição **0** do vetor
  - **comprimento(vetor)** indica o número de elementos do vetor
  - **tamanho\_do\_heap(vetor)** indica o número de elementos no heap armazenado dentro do vetor
    - Ou seja, embora  $A[1..\text{comprimento}(A)]$  contenha números válidos, nenhum elemento além de  $A[\text{tamanho\_do\_heap}(A)]$  é um elemento do heap, sendo que  $\text{tamanho\_do\_heap}(A) \leq \text{comprimento}(A)$



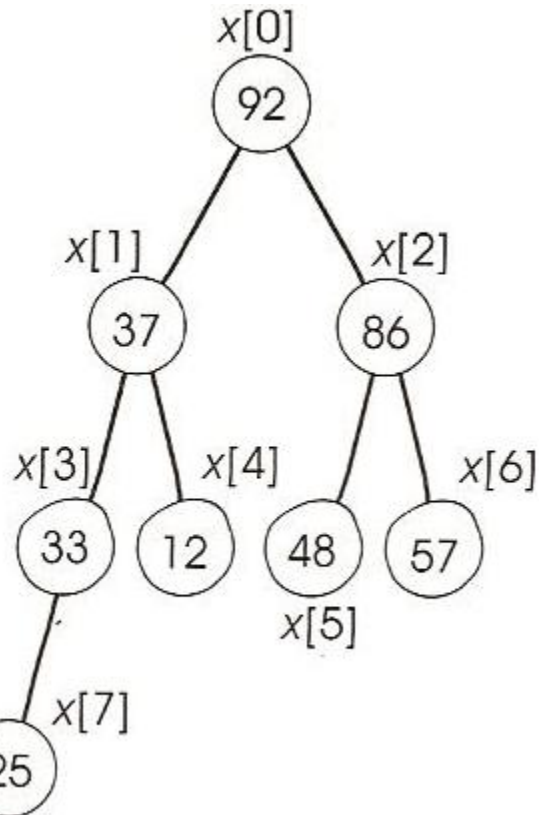
# Heap-sort

---

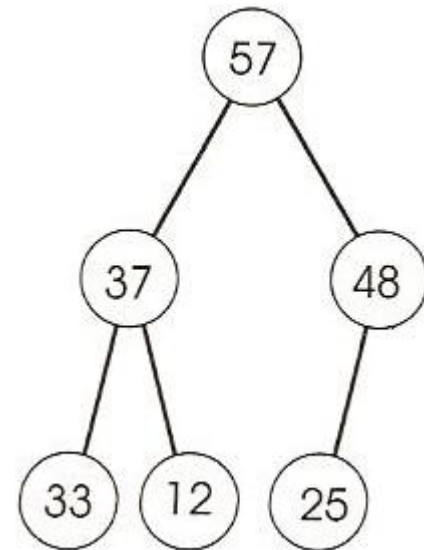
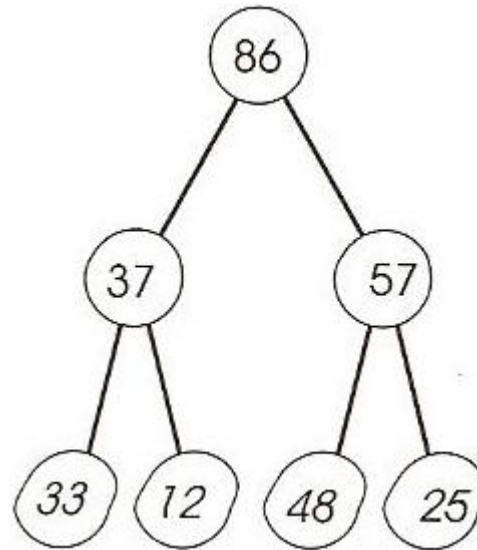
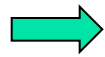
- A idéia para ordenar usando um heap é:
  - Construir um heap máximo
  - Trocar a raiz – o maior elemento – com o elemento da última posição do vetor
  - Diminuir o tamanho do heap em 1
  - Rearranjar o heap máximo, se necessário
  - Repetir o processo  $n-1$  vezes

# Heap-sort: exemplo

1) Monta-se o heap com base no vetor desordenado



3) Troca-se a raiz com o último elemento ( $x[6]$ ) e rearranja-se o heap



2) Troca-se a raiz (maior elemento) com o último elemento ( $x[7]$ ) e rearranja-se o heap



# Heap-sort

---

- O processo continua até todos os elementos terem sido incluídos no vetor de forma ordenada
- É necessário:
  - Saber construir um heap a partir de um vetor qualquer
    - Procedimento *build\_max\_heap*
  - Saber como rearranjar o heap, i.e., manter a propriedade de heap máximo
    - Procedimento *max\_heapify*



# Heap-sort

---

- Procedimento **max\_heapify**: manutenção da propriedade de heap máximo
  - Recebe como entrada um vetor A e um índice i
  - Assume que as árvores binárias com raízes nos filhos esquerdo e direito de i são heap máximos, mas que A[i] pode ser menor que seus filhos, violando a propriedade de heap máximo
  - A função do procedimento max\_heapify é deixar A[i] “escorregar” para a posição correta, de tal forma que a subárvore com raiz em i torne-se um heap máximo



# Heap-sort

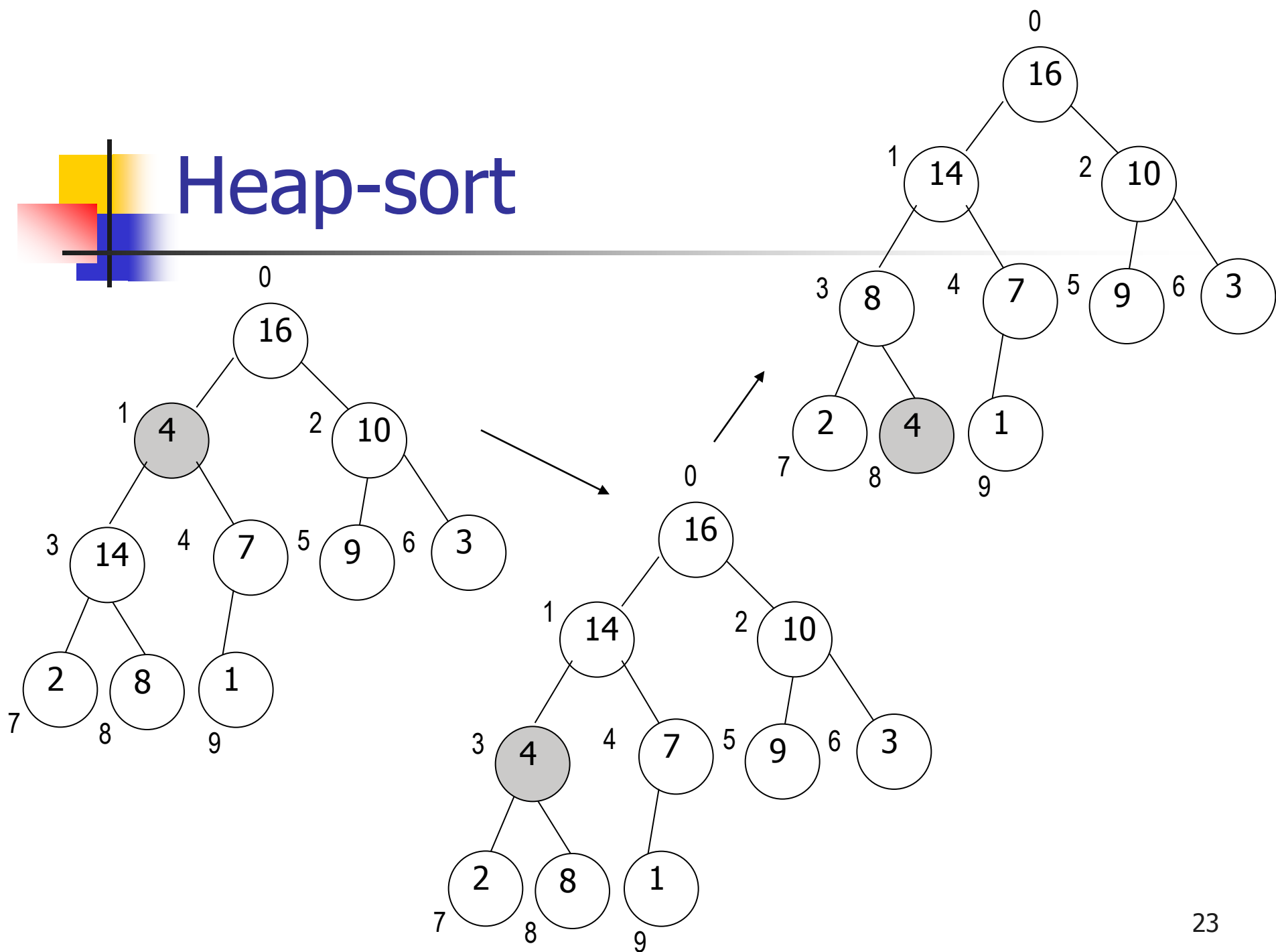
---

- Exemplo

- Chamando a função `max_heapify` para um heap hipotético

`max_heapify(A,1)`

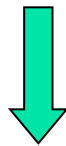
# Heap-sort



# Heap-sort

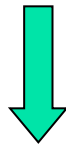
- Na realidade, trabalhando-se com o vetor A

0	1	2	3	4	5	6	7	8	9
16	4	10	14	7	9	3	2	8	1



Execução de  $\text{max\_heapify}(A,1)$

0	1	2	3	4	5	6	7	8	9
16	14	10	4	7	9	3	2	8	1



Execução recursiva de  $\text{max\_heapify}(A,3)$

0	1	2	3	4	5	6	7	8	9
16	14	10	8	7	9	3	2	4	1





# Heap-sort

---

- Implementação e análise da sub-rotina `max_heapify`



# Heap-sort

---

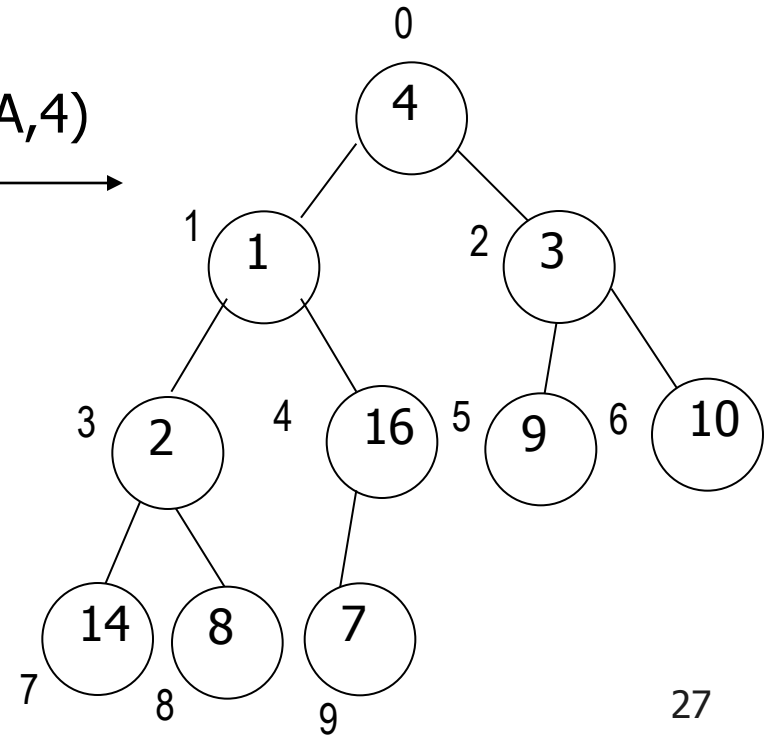
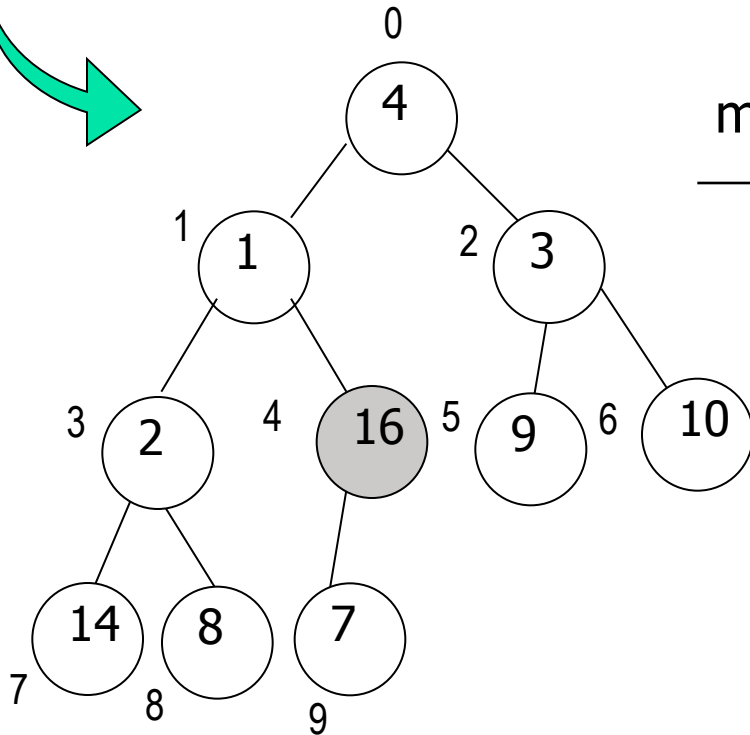
- Lembrete: as folhas do heap começam na posição  $n/2+1$
- Procedimento `build_max_heap`
  - Percorre de forma ascendente os primeiros  $n/2$  nós (que não são folhas) e executa o procedimento `max_heapify`
  - A cada chamada do `max_heapify` para um nó, as duas árvores com raiz neste nó tornam-se heaps máximos
  - Ao chamar o `max_heapify` para a raiz, o heap máximo completo é obtido

# Heap-sort

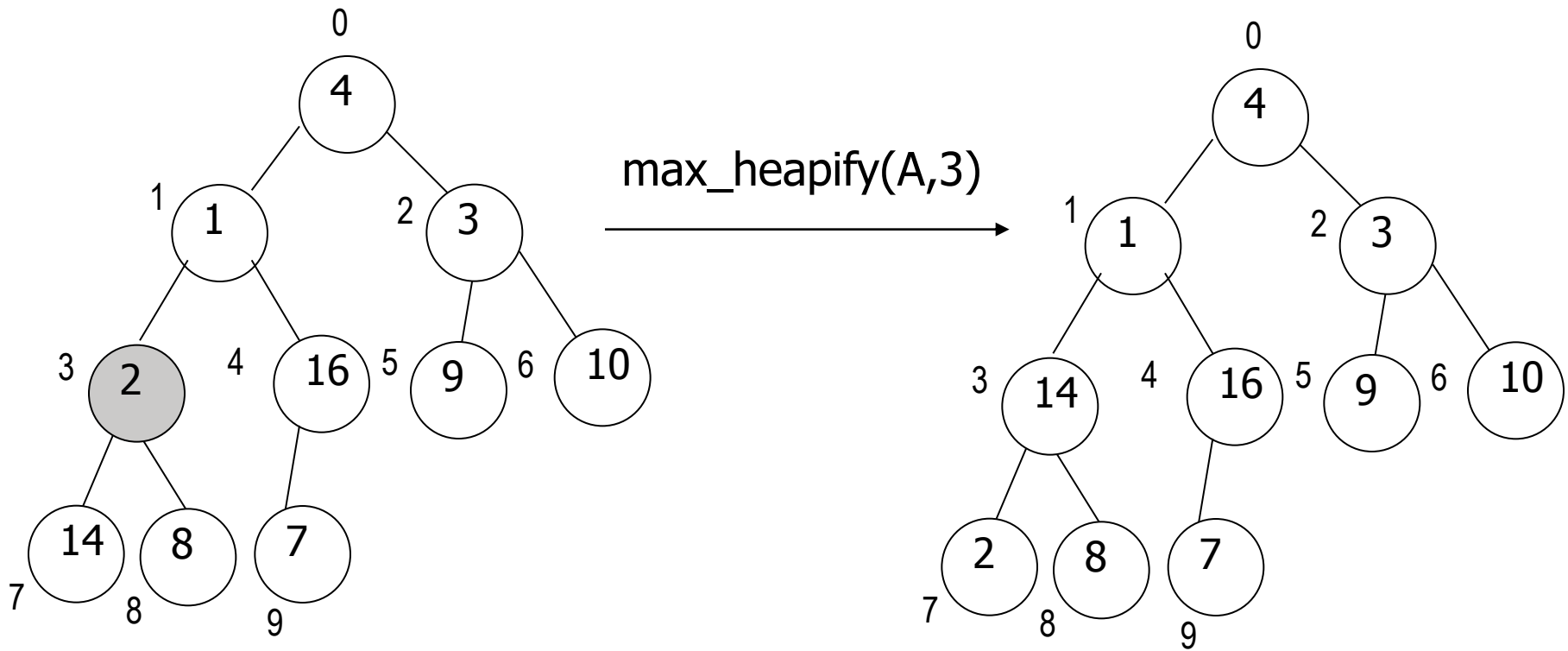
0	1	2	3	4	5	6	7	8	9
4	1	3	2	16	9	10	14	8	7

$n/2=4$

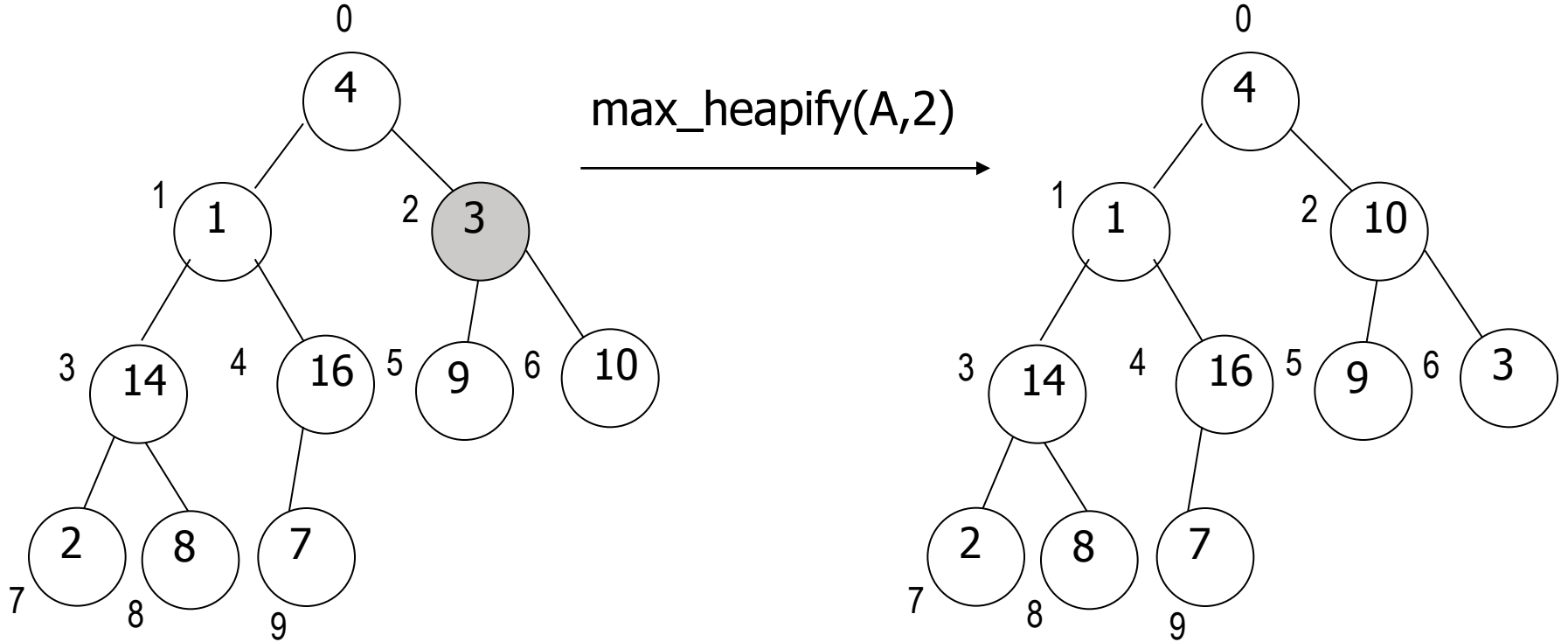
$\text{max\_heapify}(A,4)$



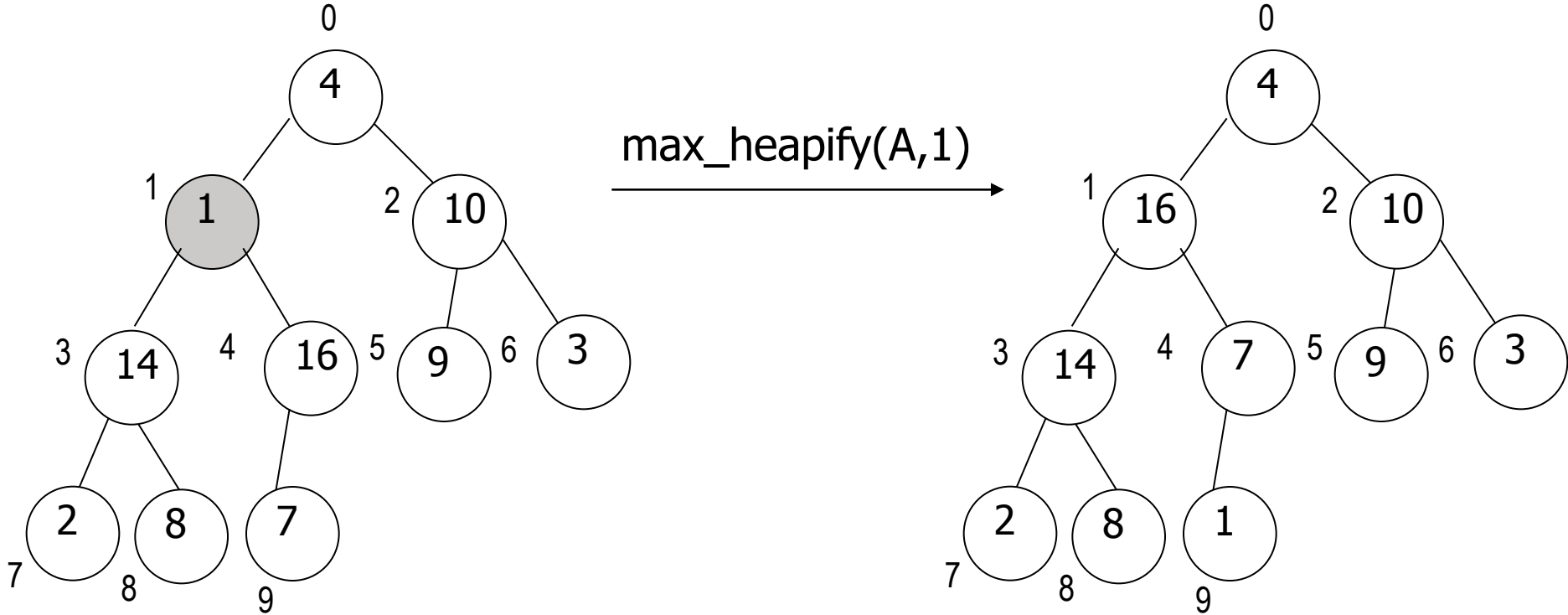
# Heap-sort



# Heap-sort

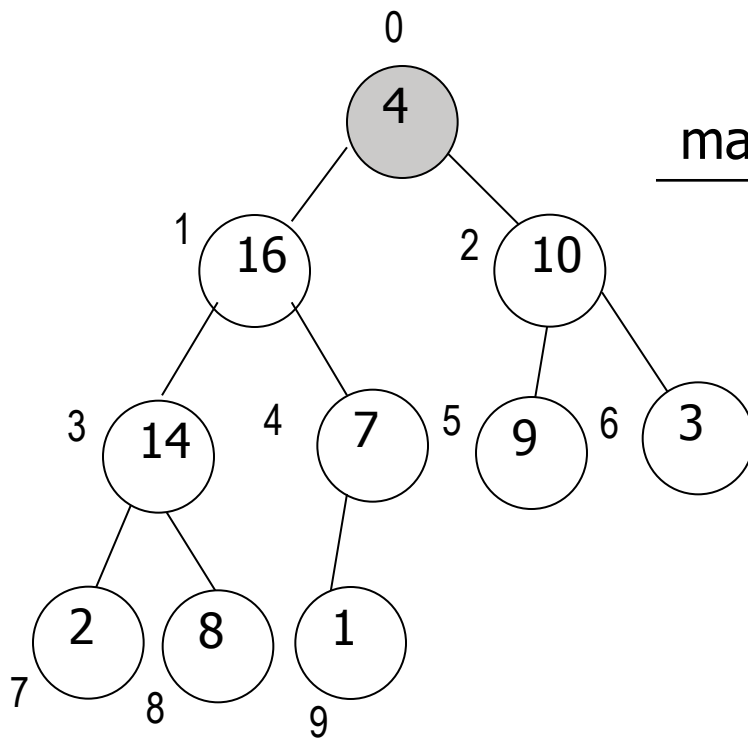


# Heap-sort

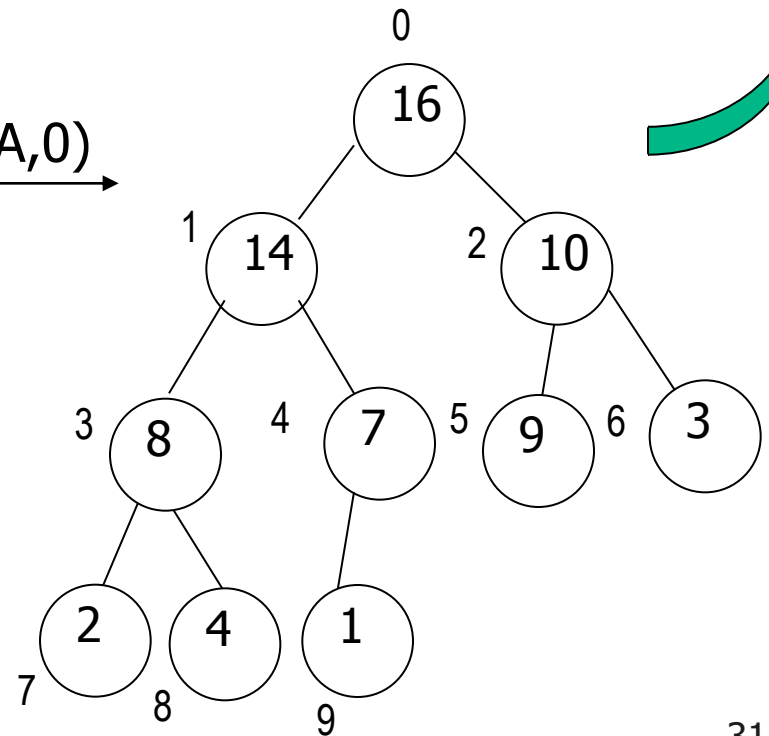


# Heap-sort

0	1	2	3	4	5	6	7	8	9
16	14	10	8	7	9	3	2	4	1



$\text{max\_heapify}(A, 0)$





# Heap-sort

---

- Implementação e análise da sub-rotina `build_max_heap`





# Heap-sort

---

- Retomando...

- Procedimento **heap-sort**

1. Construir um heap máximo (via **build\_max\_heap**)
2. Trocar a raiz – o maior elemento – com o elemento da última posição do vetor
3. Diminuir o tamanho do heap em 1
4. Rearranjar o heap máximo, se necessário (via **max\_heapify**)
5. Repetir o processo  $n-1$  vezes



# Heap-sort

---

- Dado o vetor:

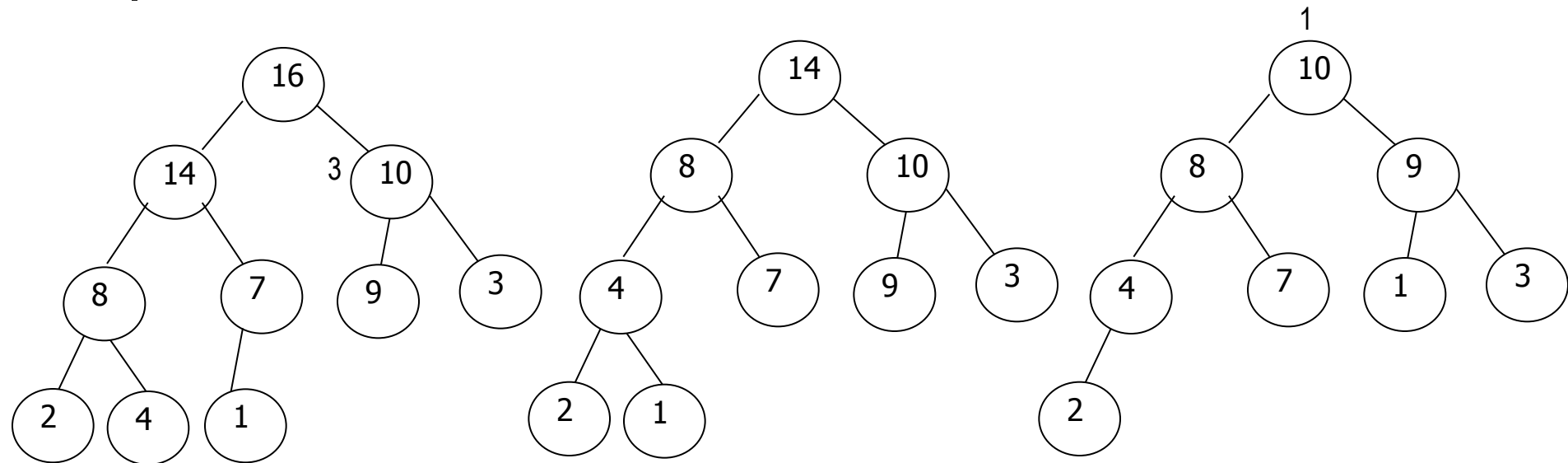
0	1	2	3	4	5	6	7	8	9
4	1	3	2	16	9	10	14	8	7

- Chamar `bulid_max_heap` e obter:

0	1	2	3	4	5	6	7	8	9
16	14	10	8	7	9	3	2	4	1

- Executar os passos de 2 a  $4n - 1$  vezes

# Heap-sort



...

0	1	2	3	4	5	6	7	8	9
1	2	3	4	7	8	9	10	14	16

Vetor ordenado!



# Heap-sort

---

- Implementação e análise da sub-rotina heap-sort



# Heap-sort

---

- O método é  $O(n \log(n))$ , sendo eficiente mesmo quando o vetor já está ordenado
  - $n-1$  chamadas a `max_heapify`, de  $O(\log(n))$ 
    - `build_max_heap` é  $O(n)$