



Métodos de Ordenação

SCC-214 Projeto de Algoritmos

Prof. Thiago A. S. Pardo



Ordenação por Seleção

- Idéia básica: os elementos são selecionados e dispostos em suas posições corretas finais
 - Seleção direta (ou simples), ou classificação de deslocamento descendente
 - Um dos primeiros métodos que alunos implementam
 - Heap-sort, ou método do monte



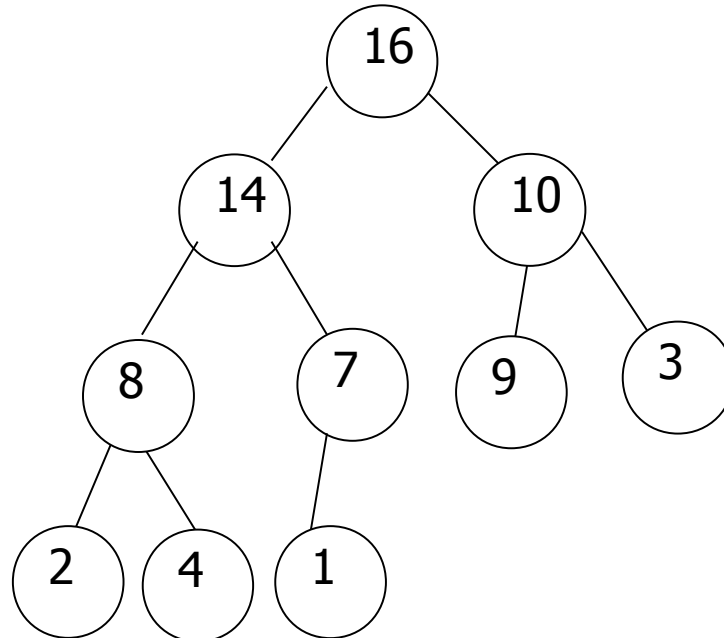
Heap-sort

- Utiliza 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 variáveis dinâmicas”



Heap-sort

- Um heap é uma estrutura de dados em que há uma ordenação entre elementos: representação via árvore binária





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): pai \geq filhos, sendo que a raiz é o maior elemento
 - *Propriedade de heap máximo*
 - **Heap mínimo** (ou heap ascendente): pai \leq filhos, sendo que a raiz é o menor elemento
 - *Propriedade de heap mínimo*



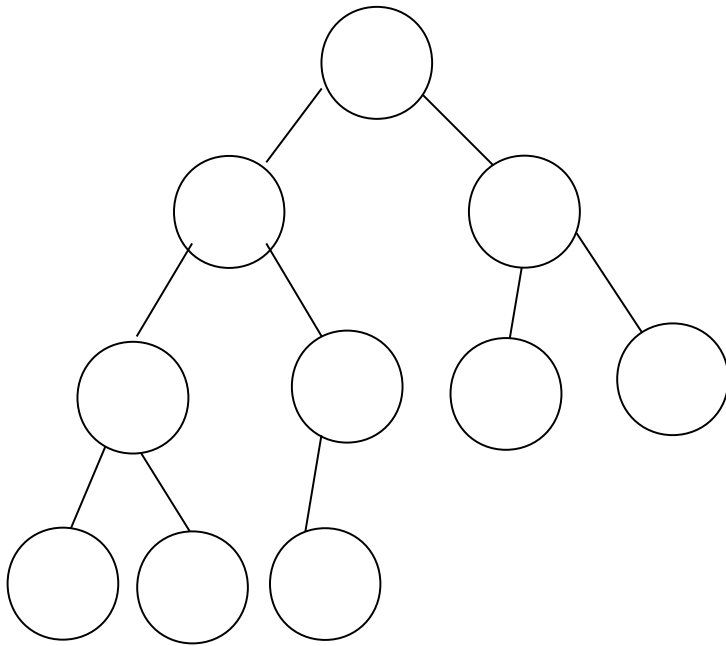
Heap-sort

- Um heap observa conceitos de **ordem** e de **forma**
 - **Forma**: a árvore binária tem seus nós-folha, no máximo, em dois níveis, sendo que as folhas devem estar o mais à esquerda possível

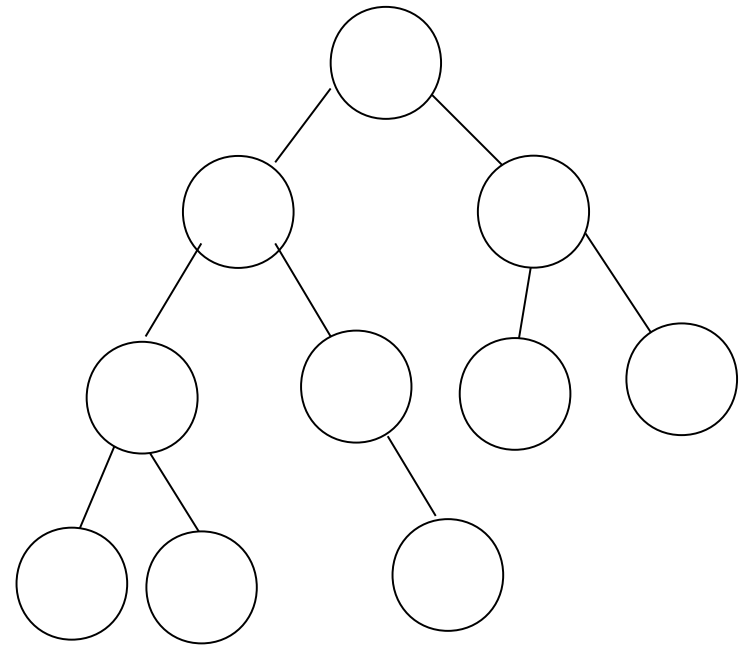


Heap-sort

- Exemplos



OK

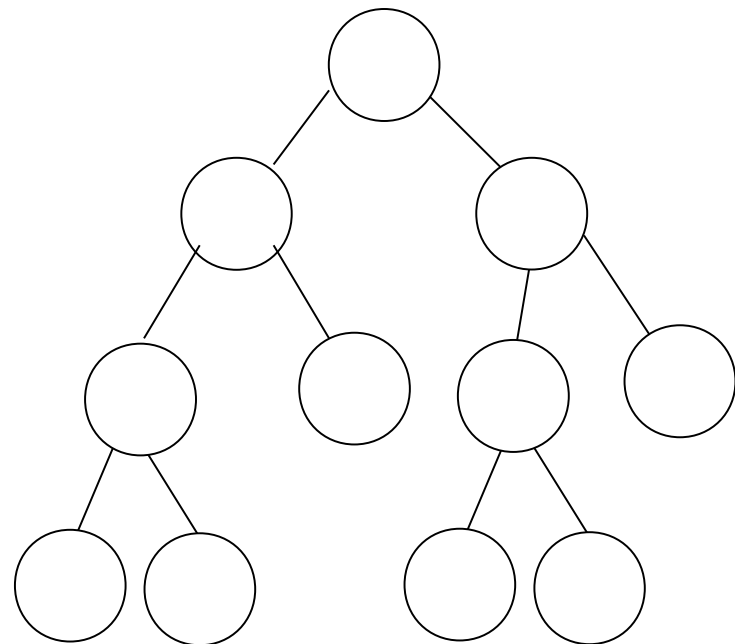
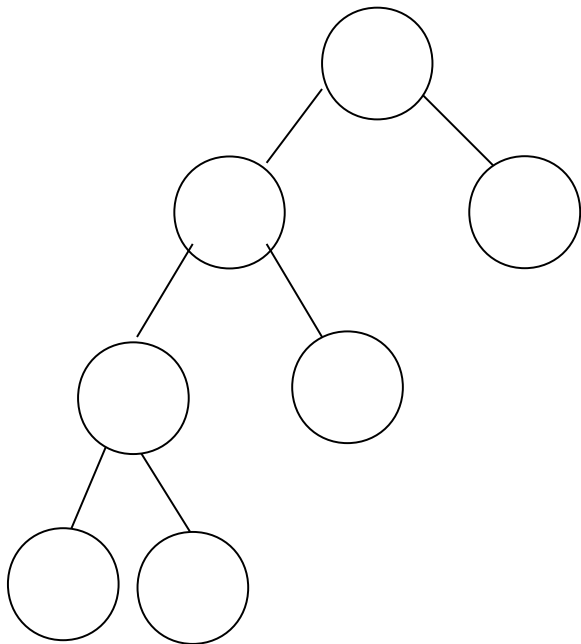


Não!



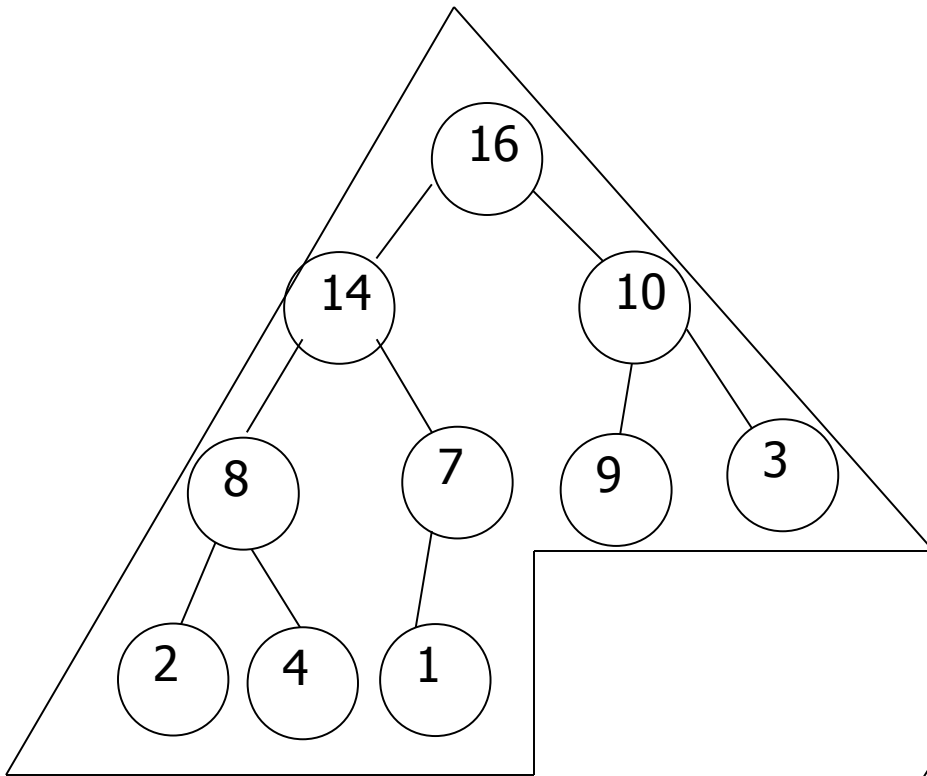
Heap-sort

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

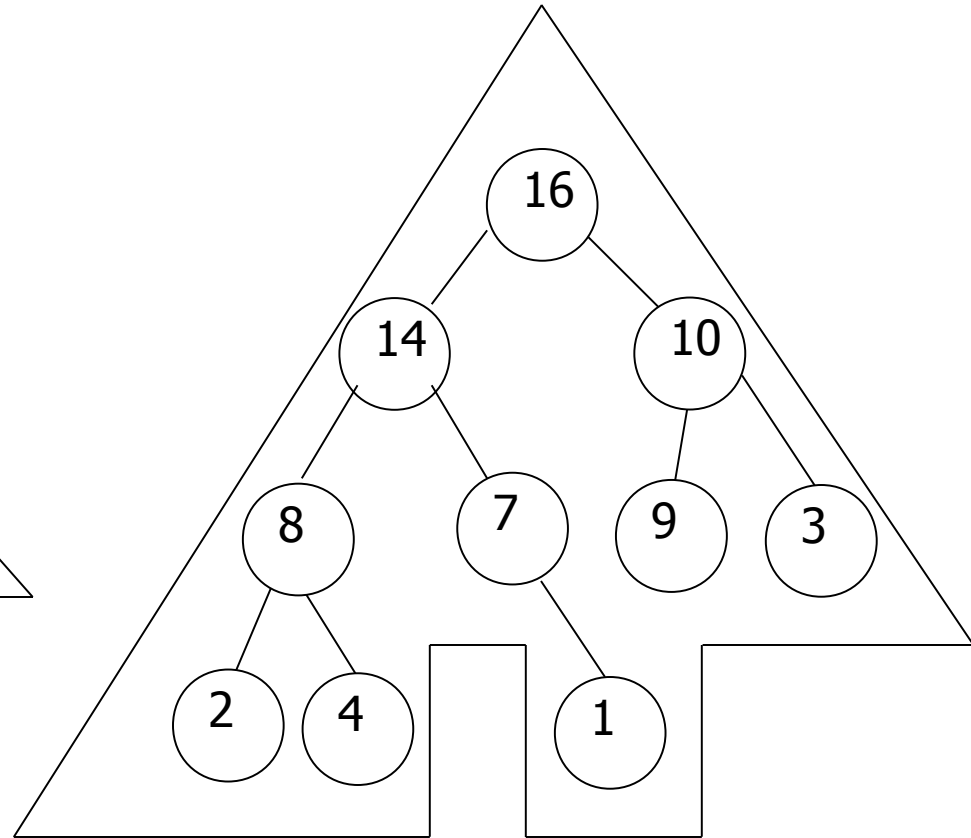




Heap-sort

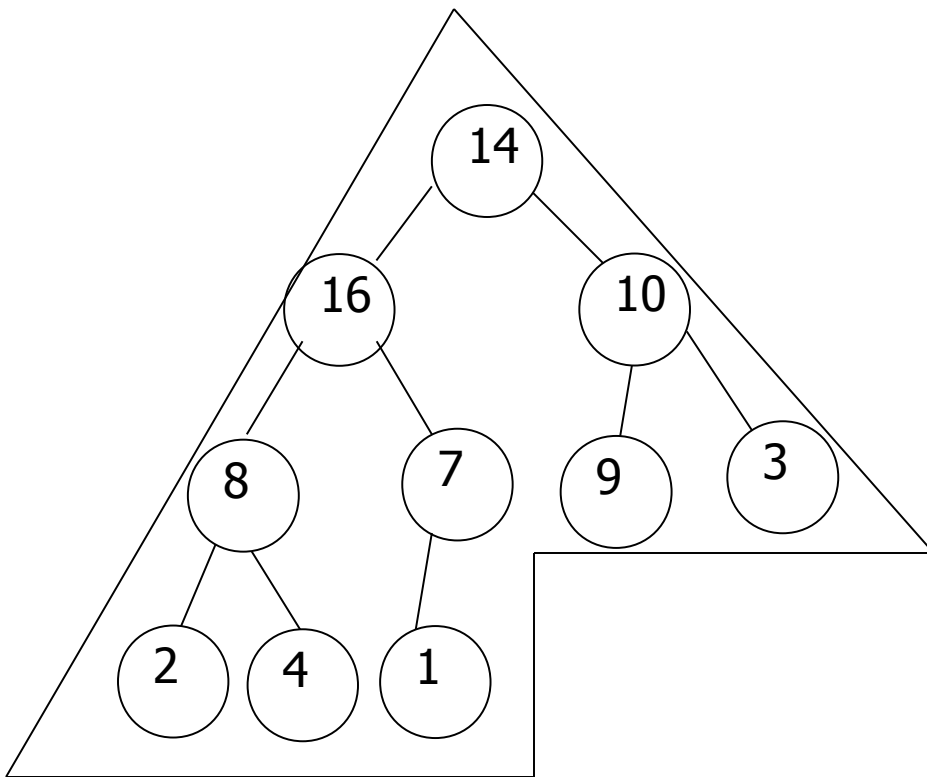


É um heap máximo

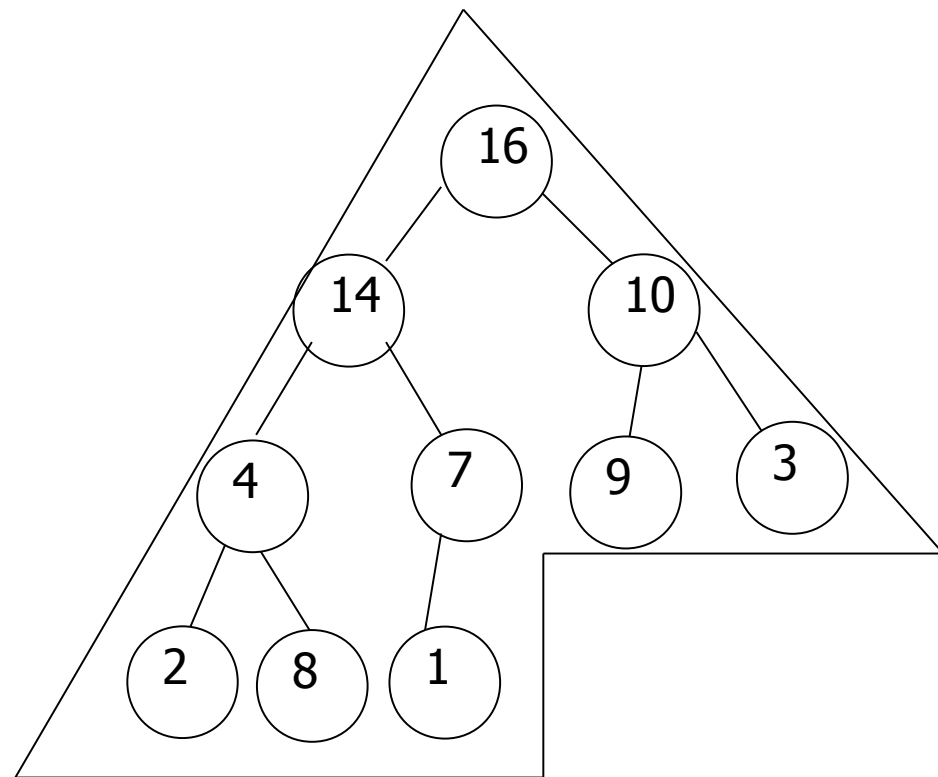


Não é um heap máximo ₉

Heap-sort



Não é um heap máximo



Não é um heap máximo

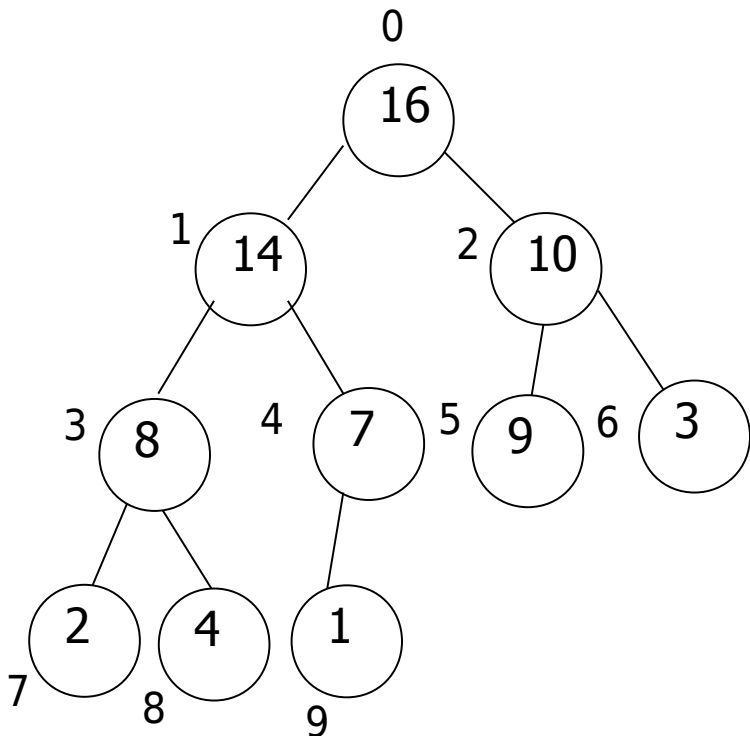


Heap-sort

- Pergunta
 - Como seria um heap mínimo?

Heap-sort

- Um heap pode ser representado por um vetor

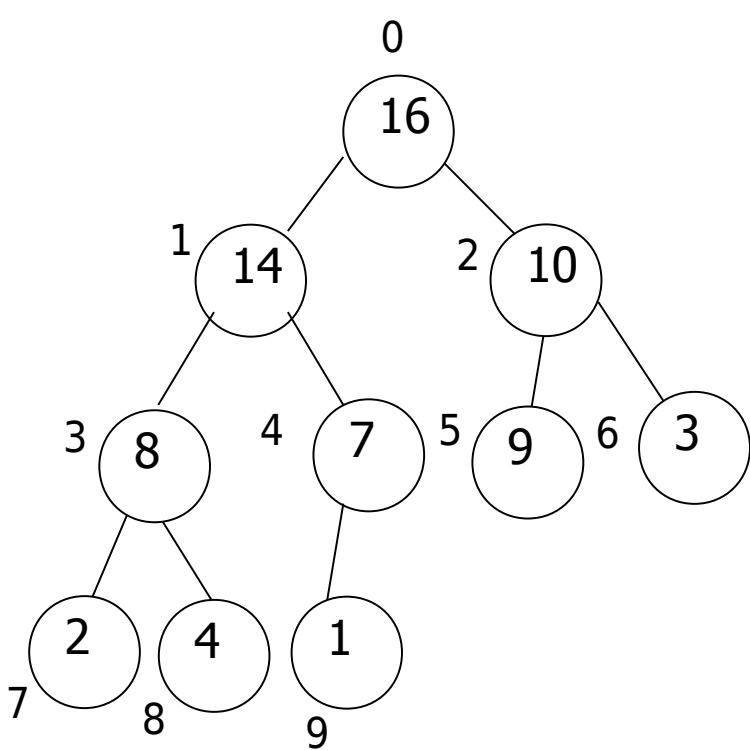


0	1	2	3	4	5	6	7	8	9
16	14	10	8	7	9	3	2	4	1



Heap-sort

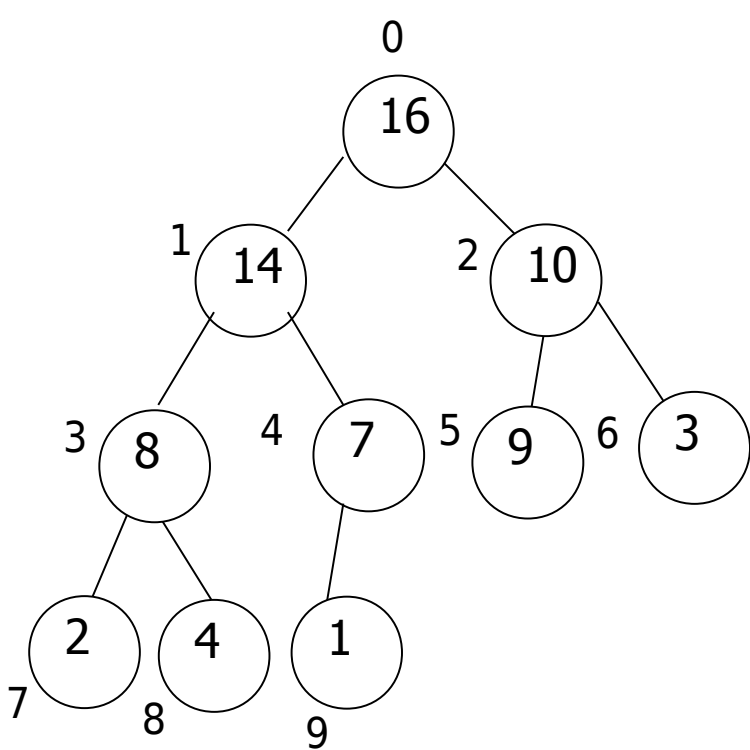
- Como acessar os elementos (pai e filhos de cada nó) no heap?



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

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

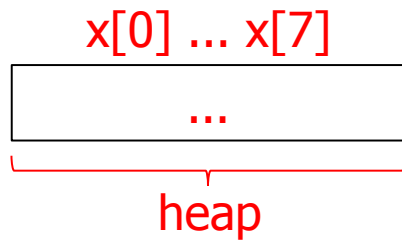
- Assume-se que:
 - A raiz está sempre na posição **0** do vetor
 - `comprimento(vetor)` indica o número de elementos do vetor
 - `tamanho_do_heap(vetor)` indica o número de elementos no heap armazenado dentro do vetor
 - Ou seja, embora $A[1..\text{comprimento}(A)]$ contenha números válidos, nenhum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



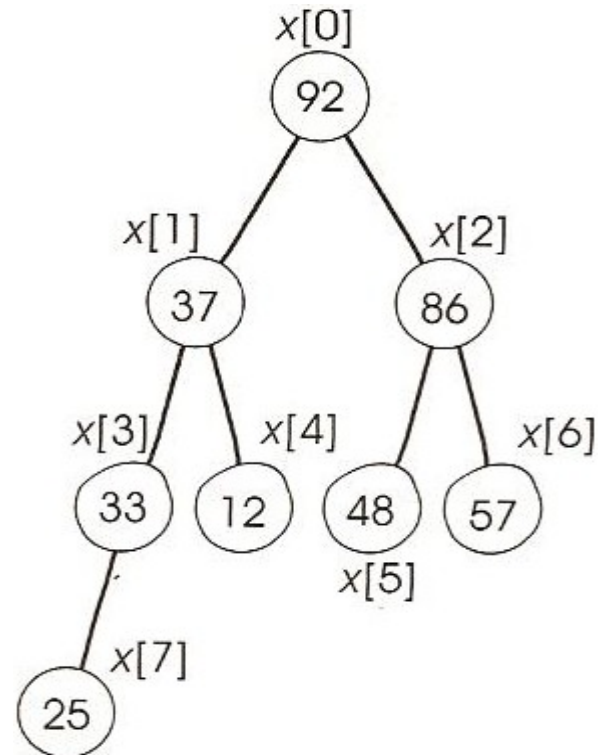
Heap-sort

- 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 (agora menor), se necessário
 - Repetir o processo $n-1$ vezes

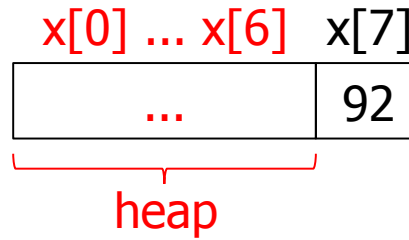
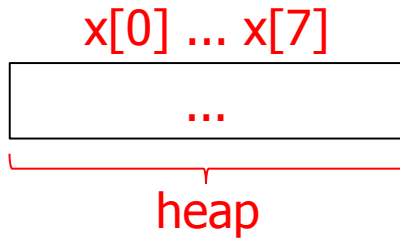
Heap-sort: exemplo



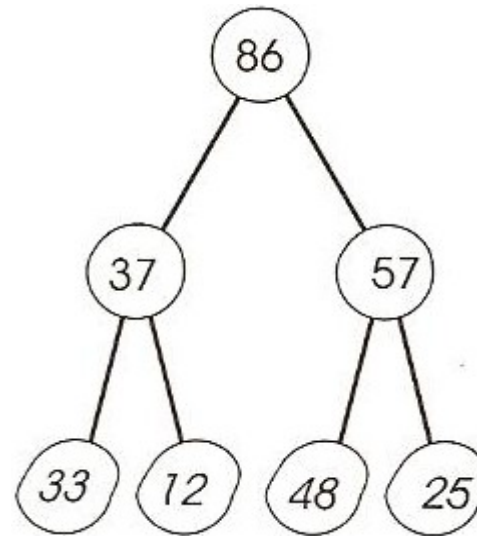
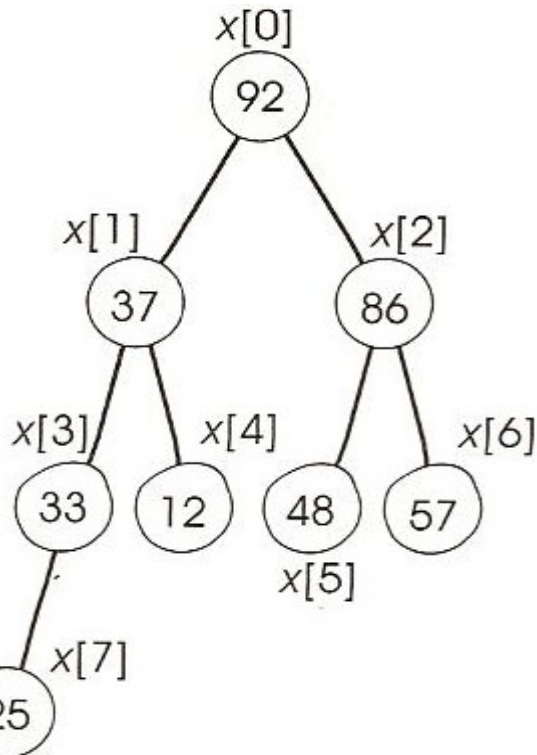
1) Monta-se o heap com base no vetor desordenado



Heap-sort: exemplo

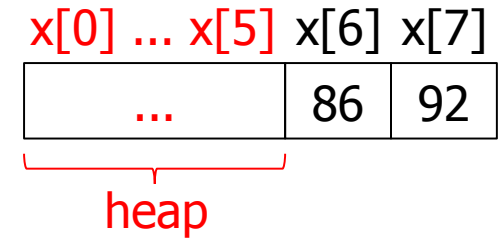
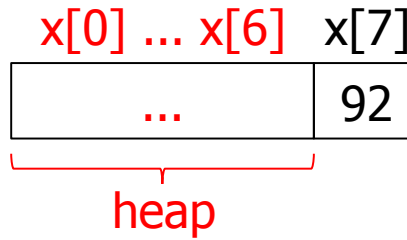
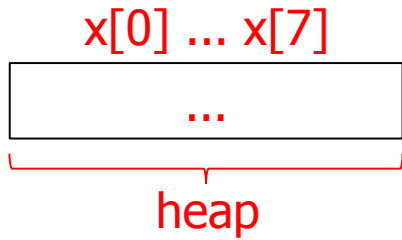


1) Monta-se o heap com base no vetor desordenado



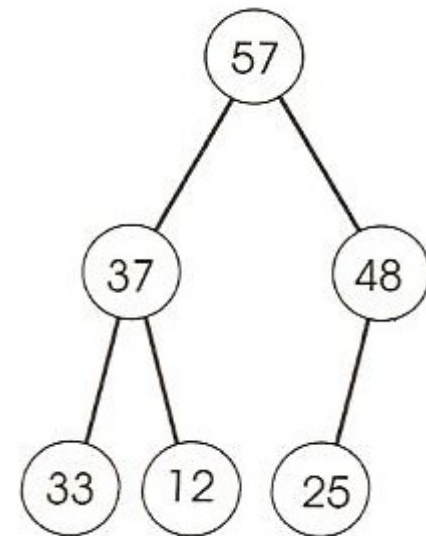
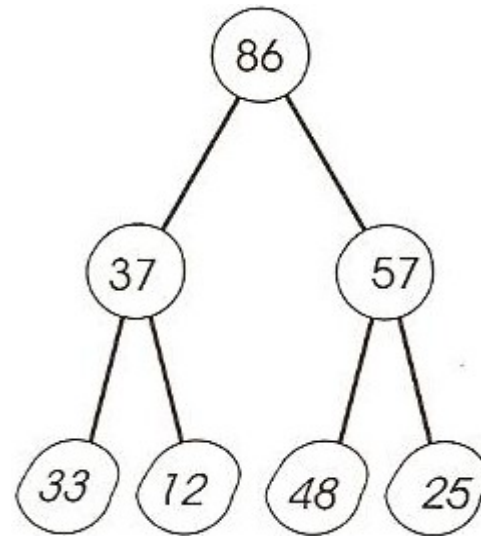
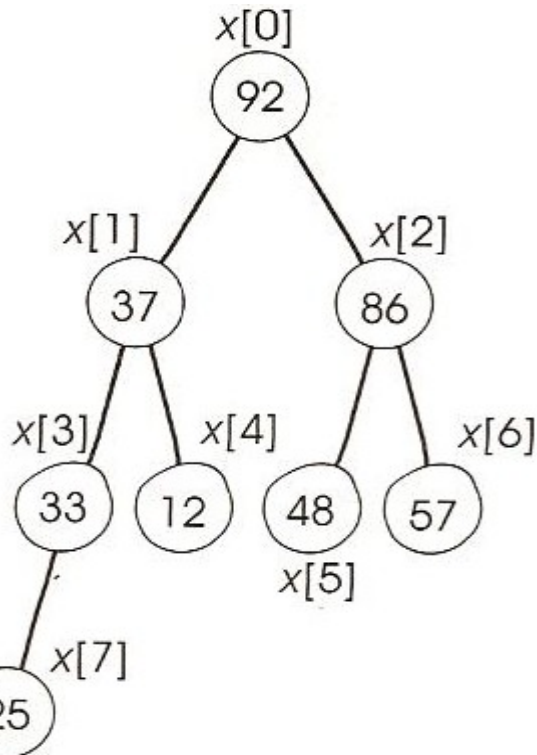
2) Troca-se a raiz (maior elemento) com o último elemento ($x[7]$) e rearranja-se o heap

Heap-sort: exemplo



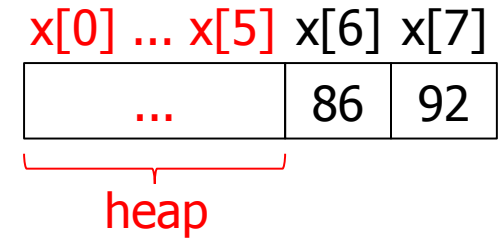
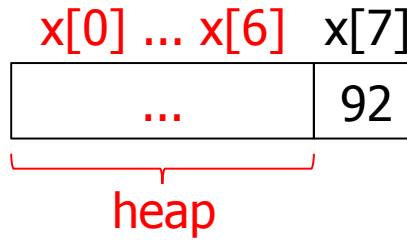
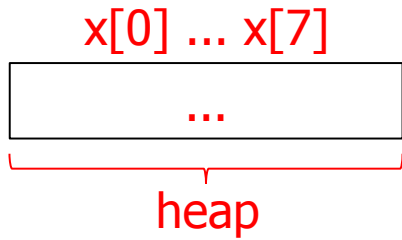
1) Monta-se o heap com base no vetor desordenado

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



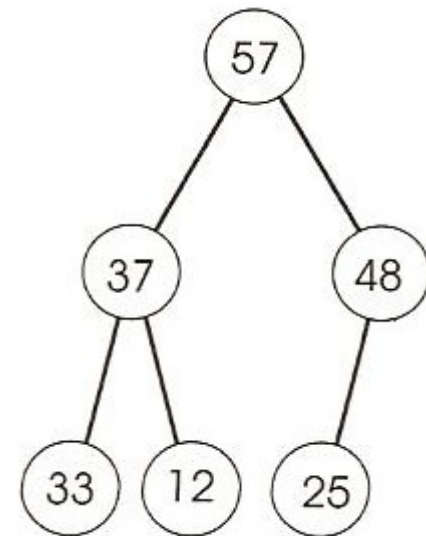
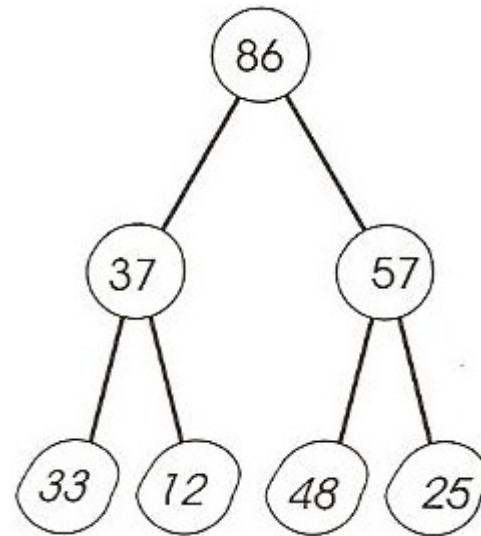
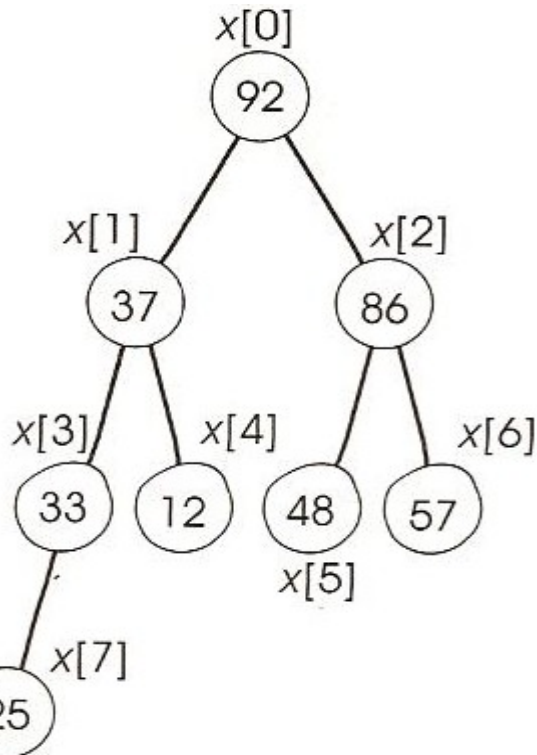
2) Troca-se a raiz (maior elemento) com o último elemento ($x[7]$) e rearranja-se o heap

Heap-sort: 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 *construir_heap*
 - Saber como rearranjar o heap, i.e., manter a propriedade de heap máximo
 - Procedimento *rearranjar_heap*



Heap-sort

- Procedimento *rearranjar_heap*: 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 *rearranjar_heap* é deixar A[i] “escorregar” para a posição correta, de tal forma que a subárvore com raiz em i torne-se um heap máximo

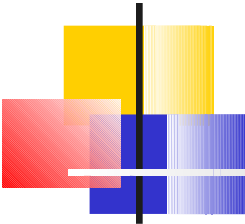


Heap-sort

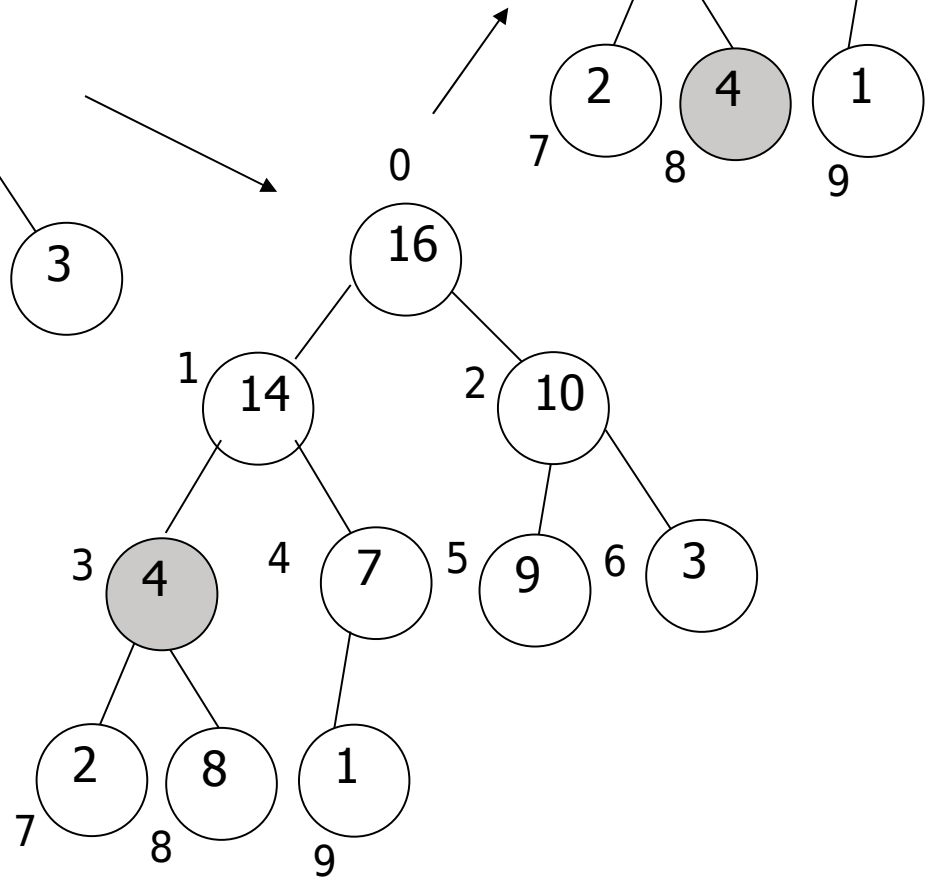
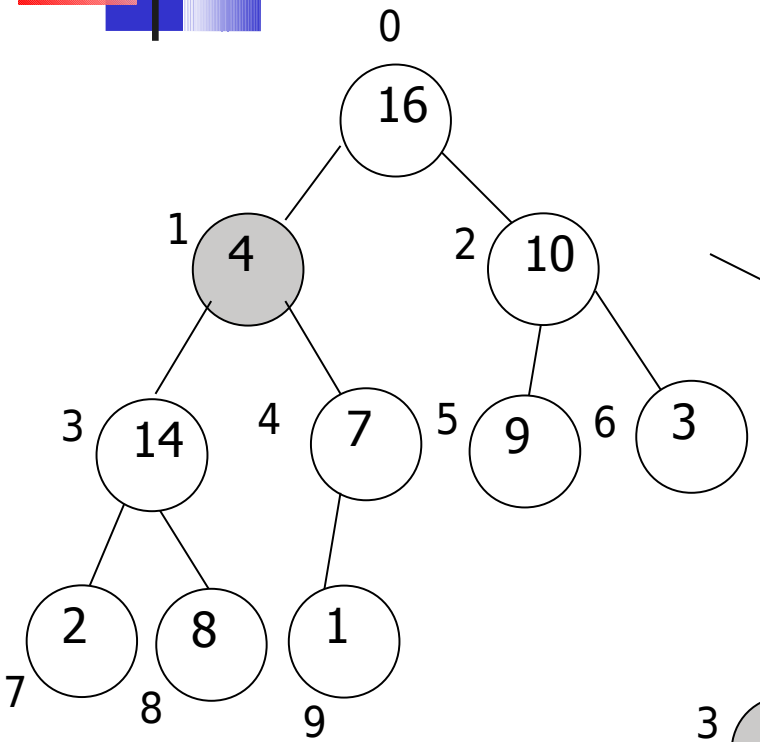
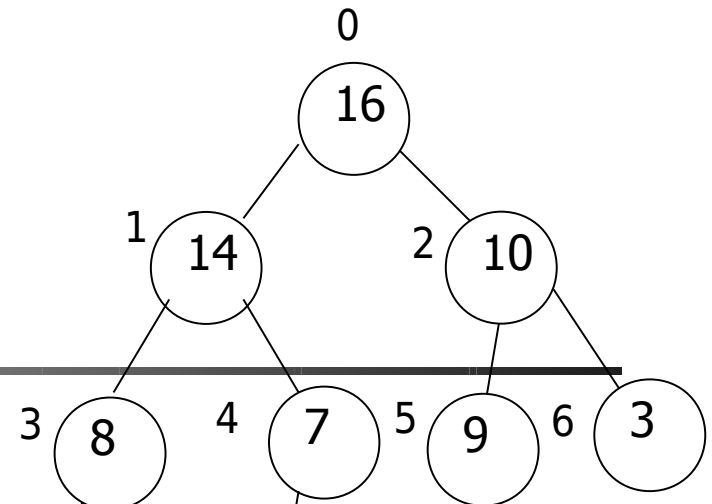
- Exemplo

- Chamando a função *rearranjar_heap* para um heap hipotético

rearranjar_heap(A,1)



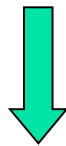
Heap-sort



Heap-sort

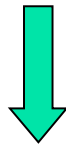
- Na realidade, trabalhando-se com o vetor A

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



Execução de *rearranjar_heap(A,1)*

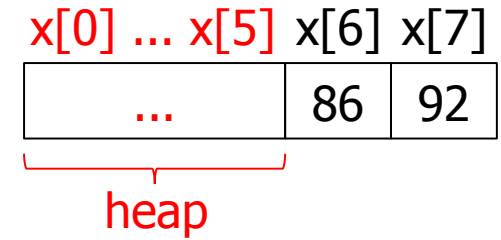
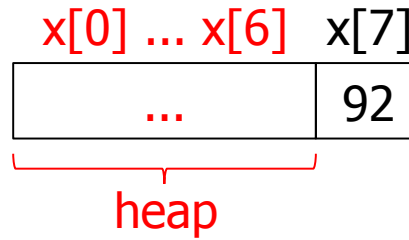
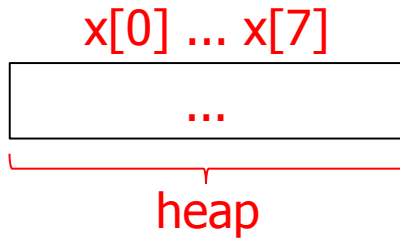
0	1	2	3	4	5	6	7	8	9
16	14	10	4	7	9	3	2	8	1



Execução recursiva de *rearranjar_heap(A,3)*

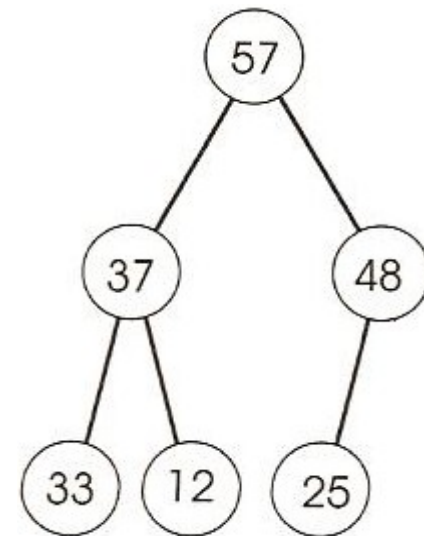
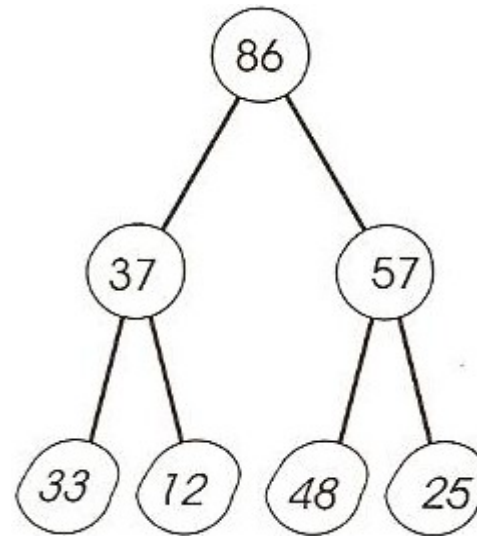
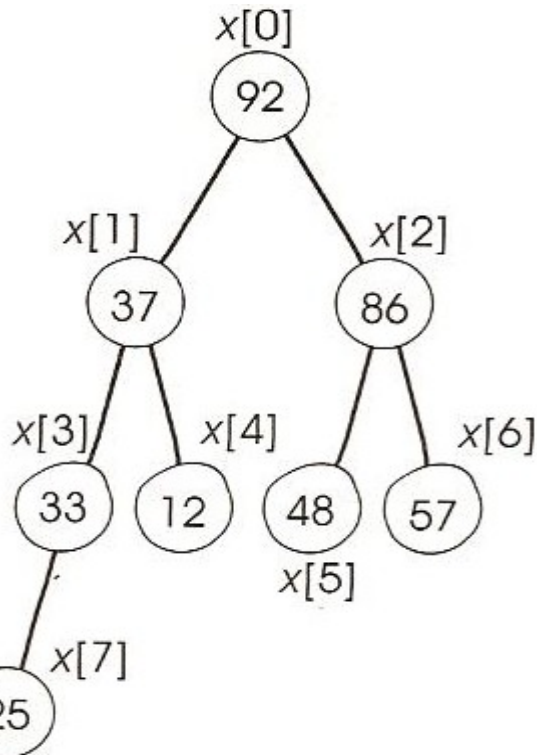
0	1	2	3	4	5	6	7	8	9
16	14	10	8	7	9	3	2	4	1

Como acontece?



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

- Implementação e análise da sub-rotina *rearranjar_heap*

```
void rearranjar_heap(int v[], int i, int tamanho_do_heap)
```

→ v = vetor

→ i = nó a partir do qual é necessário rearranjar



Heap-sort

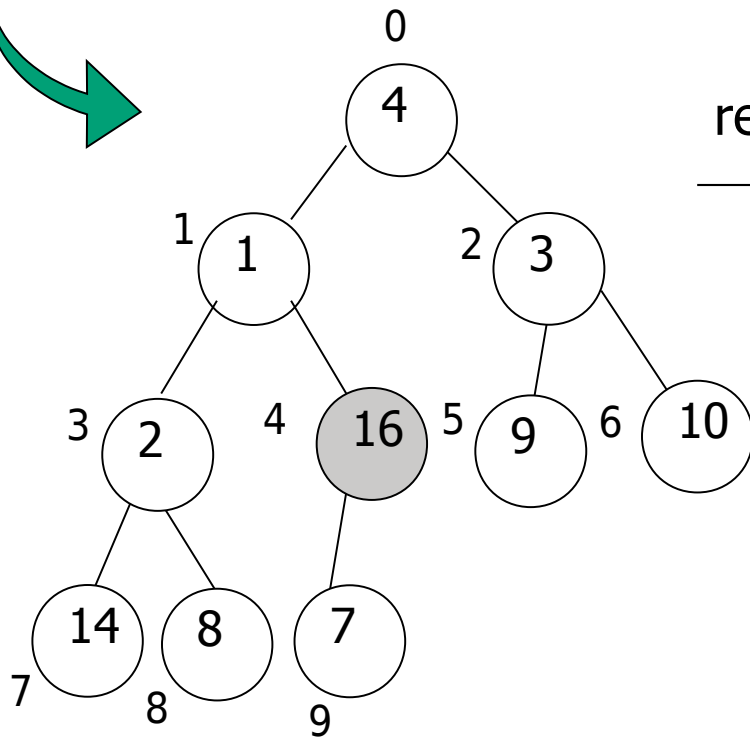
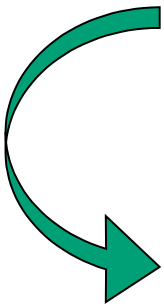
- Lembrete: as folhas do heap começam na posição $n/2$
- Procedimento *construir_heap*
 - Percorre de forma ascendente os primeiros $n/2 - 1$ nós (que não são folhas) e executa o procedimento *rearranjar_heap*
 - A cada chamada do *rearranjar_heap* para um nó, as duas árvores com raiz neste nó tornam-se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



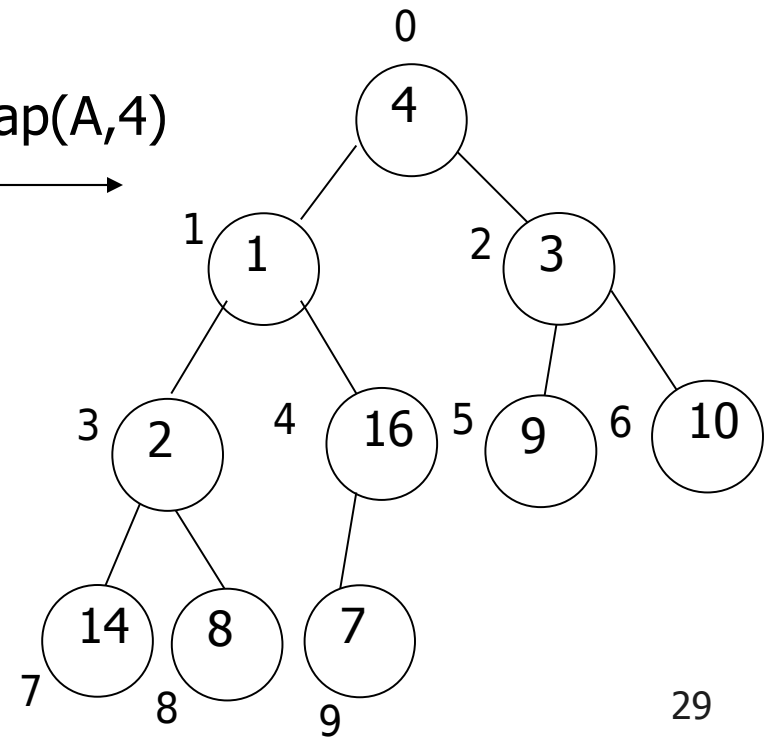
Heap-sort

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

$$n/2 - 1 = 4$$

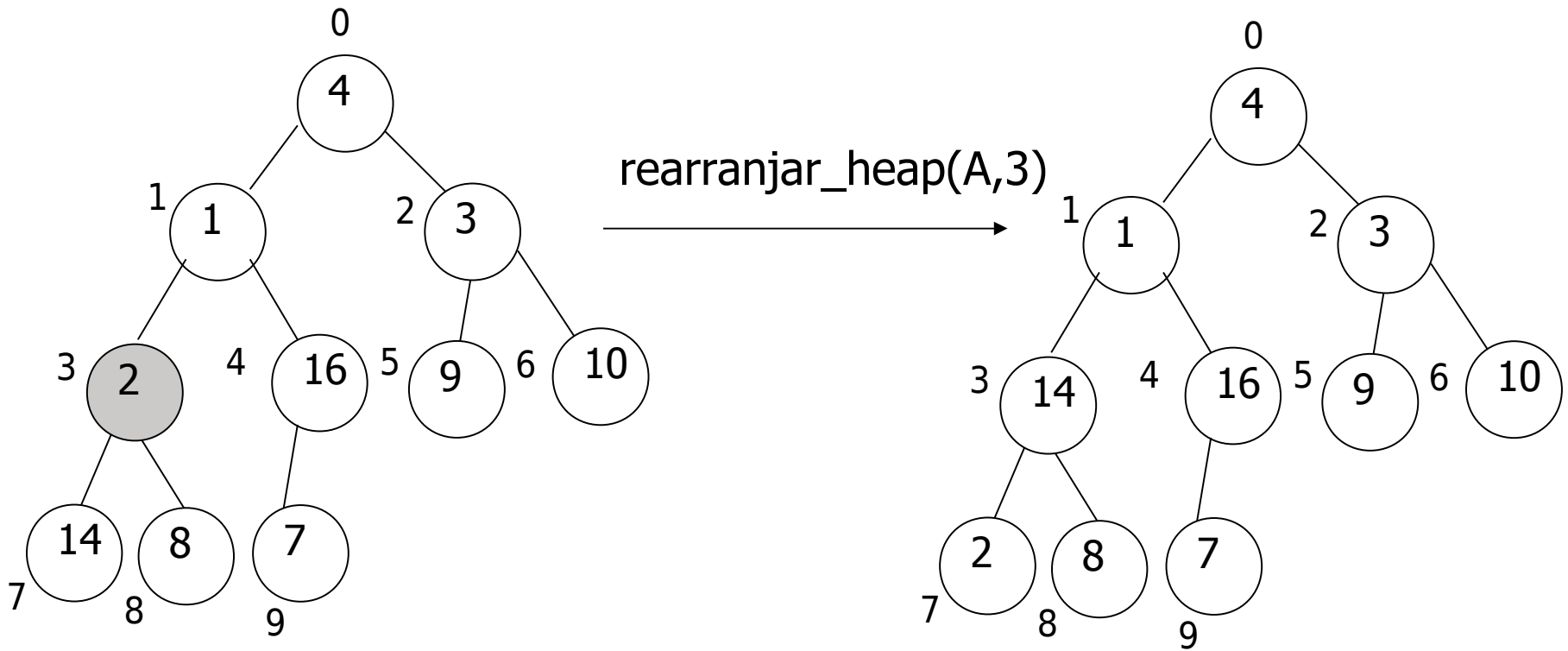


rearranjar_heap(A,4)



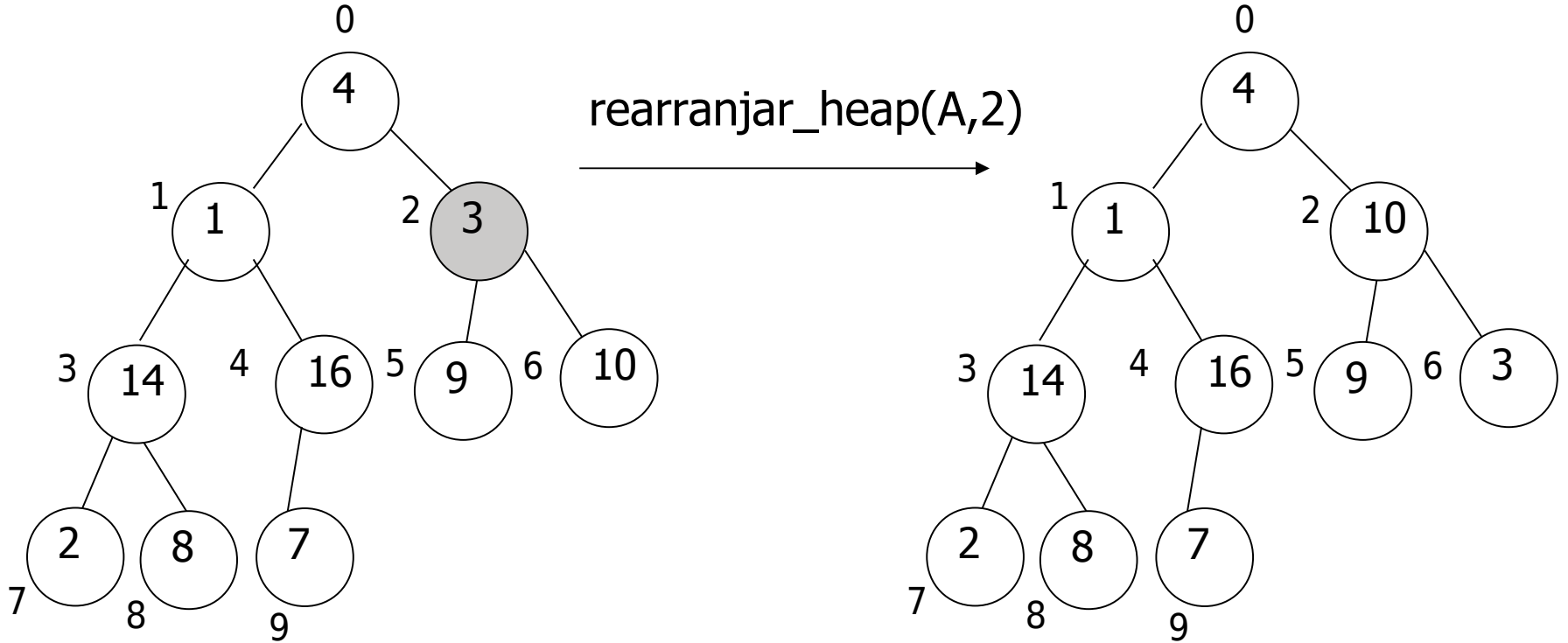


Heap-sort



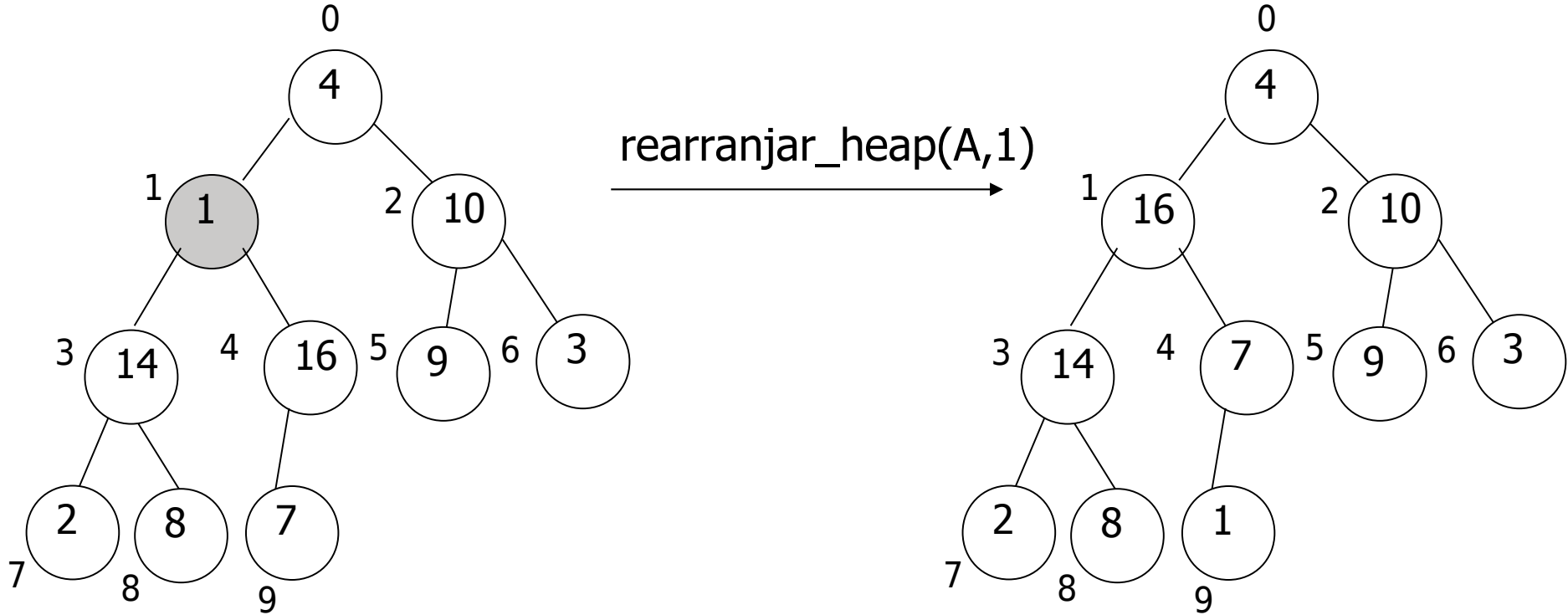


Heap-sort



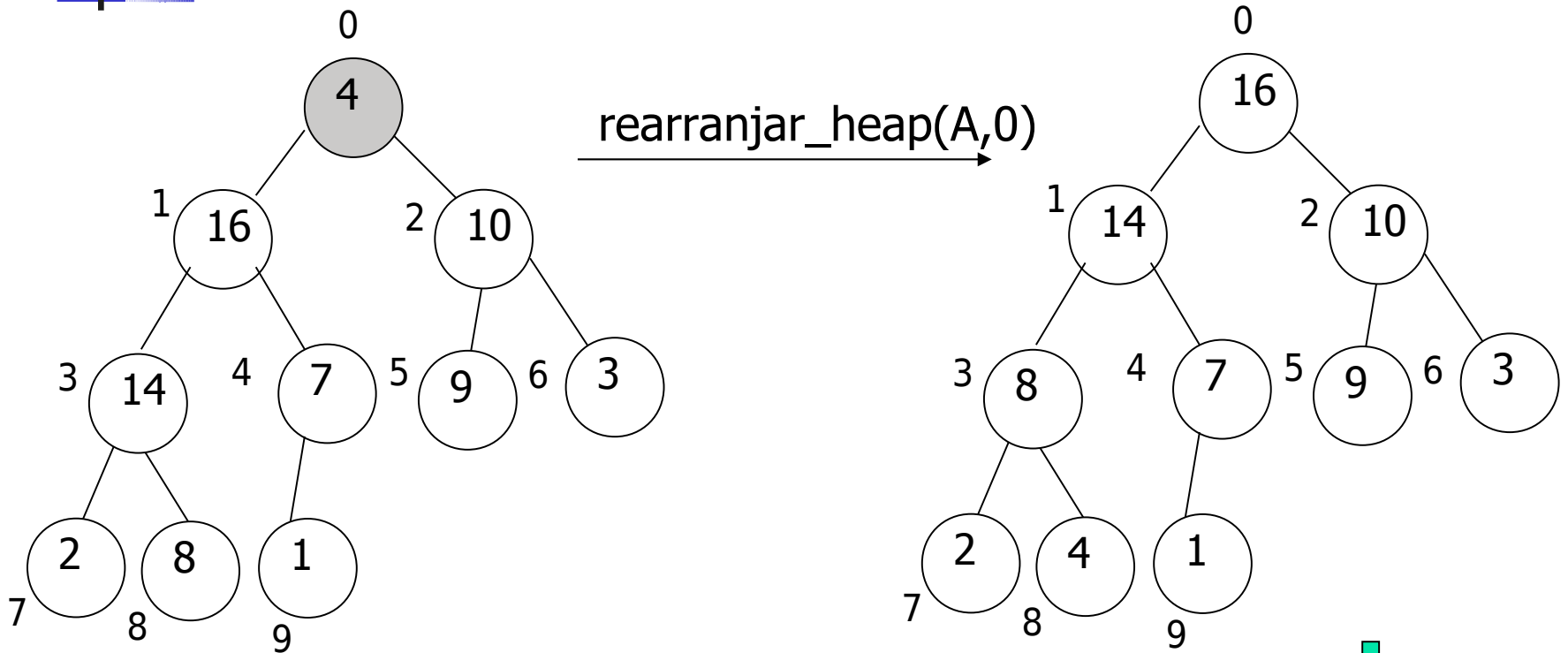


Heap-sort

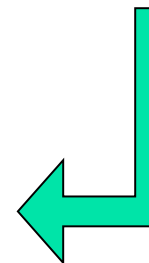




Heap-sort



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





Heap-sort

- Implementação e análise da sub-rotina *construir_heap*

```
void construir_heap(int v[], int n)
```



Heap-sort

- Retomando...

- Procedimento **heap-sort**

1. Construir um heap máximo (via **construir_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 **rearranjar_heap**)
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

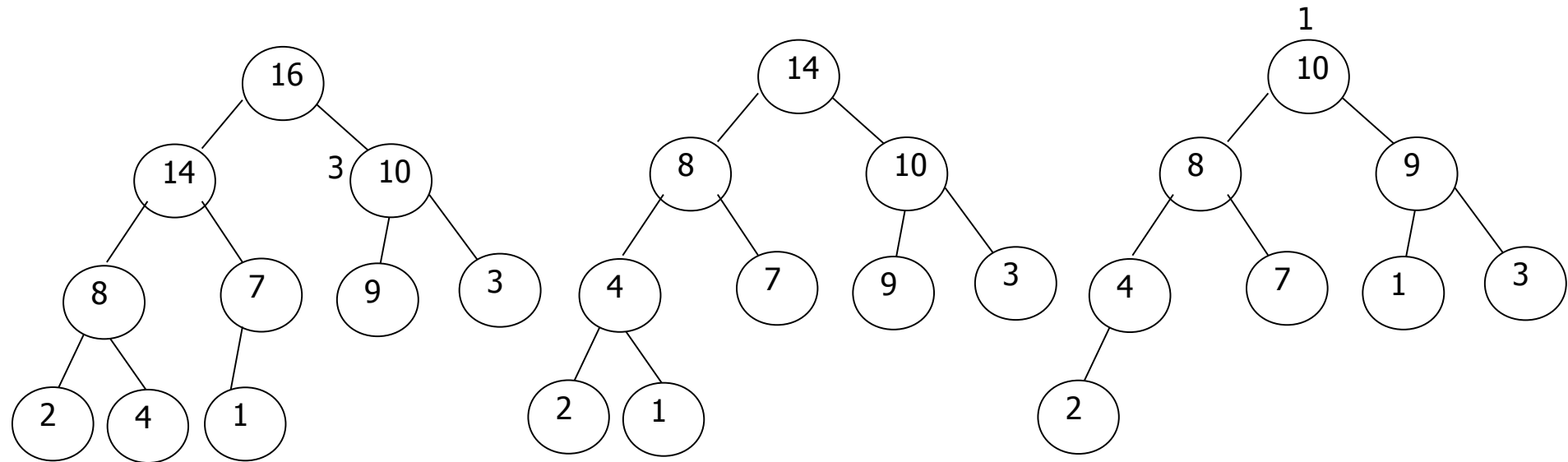
- Chamar `construir_heap` e obter:

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

```
void heapsort(int v[], int n)
```



Heap-sort

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



Heap-sort

- Executar o processo de ordenação completo para o vetor abaixo

(44 , 55 , 12 , 42 , 94)