

Algoritmos de Ordenação

Prof. Martín Vigil

Objetivo dos Algoritmos de Ordenação

- Seja uma sequência de objetos com alguma característica comparável (i.e, *chave*), por exemplo:
 - Sequência de pessoas, cada uma com um CPF distinto
 - Sequência de pessoas, cada uma com uma idade
- Não necessariamente todos os objetos são distintos
- Um algoritmo de ordenação ordena a sequência em ordem
 - Não-decrescente: do menor para o maior
 - Não-crescente: dos maior para os menor

Algoritmos de ordenação ditos *simples*

Fáceis de implementar
porém
ineficientes ($O(n^2)$)

Algoritmos de ordenação ditos *eficientes*

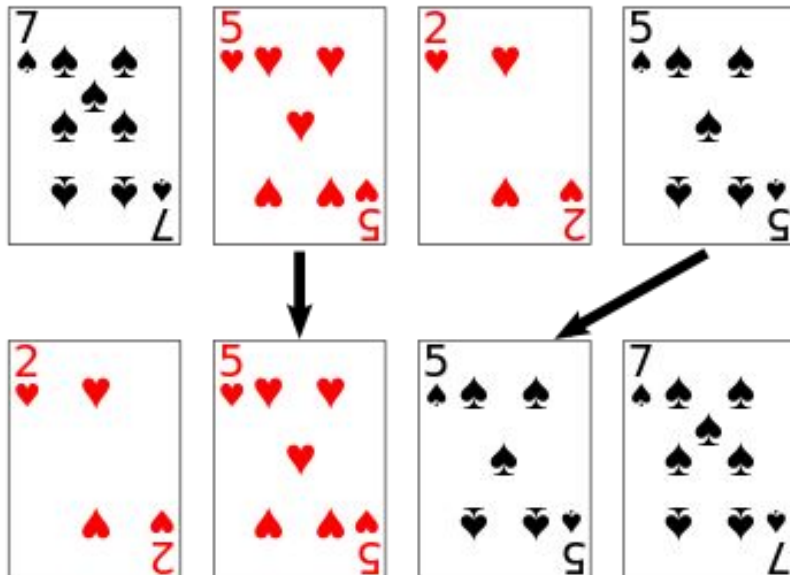
Eficientes $O(n \log n)$
porém
difíceis de implementar

Algoritmos *estáveis*

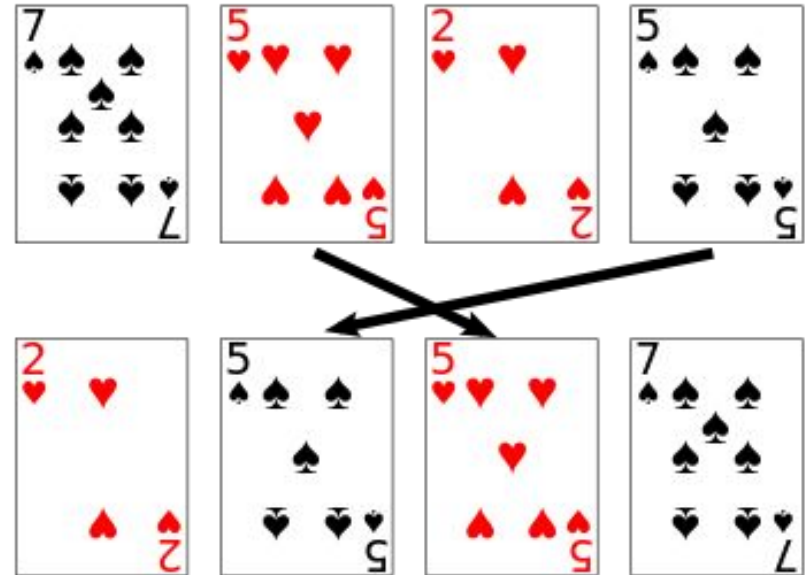
Não mudam a ordem
relativa de dois objetos
com **chaves iguais**

Estabilidade: ordem pelo valor numérico

Stable



Not stable



Seja $v = v[1], v[2], \dots, v[n]$ um vetor
de $n > 0$ inteiros

Para $1 \leq i \leq n' \leq n$, $v' = v[i], v[i+1], \dots, v[n']$
é um subvetor de v

Estados iniciais do vetor a ordenar: melhor, pior e médio casos

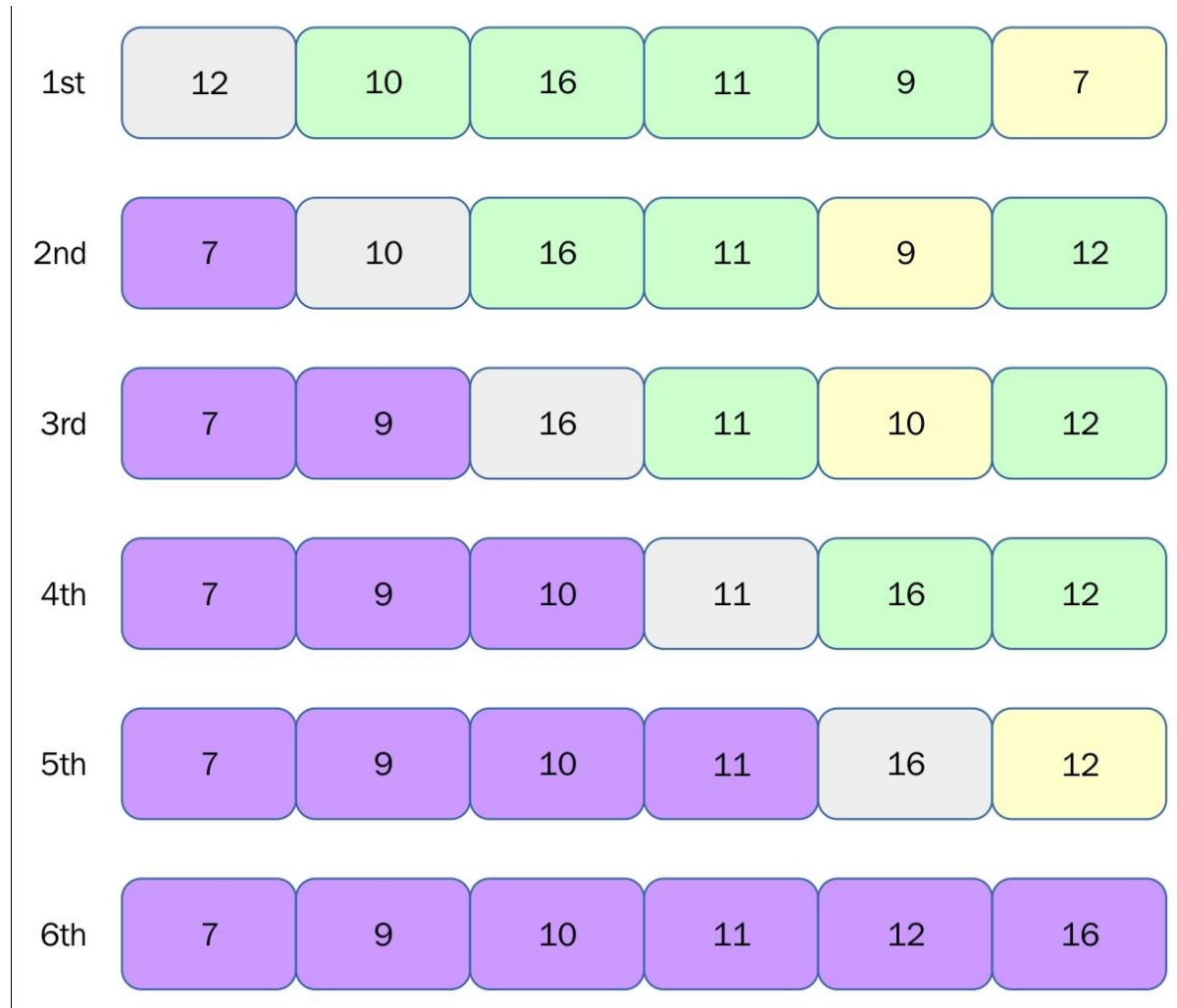


Algoritmos Simples

Algoritmo da Seleção: Ideia

Selecionar o elemento mínimo de um vetor (resp. subvetor) e movê-lo para a primeira posição do vetor (resp. subvetor)

Algoritmo da Seleção: Exemplo



Algoritmo da Seleção

```
1. função selectionSort(vetor v, tamanho n)
2.     para j=1,2,..., n-1
3.         minimo = j
4.         para k=j+1,j+2,...,n
5.             se v[k] < v[minimo]
6.                 minimo = k
7.
8.         troca(v[minimo], v[j])
```

Algoritmo da Seleção: Complexidade

```
1. função selectionSort(vetor v, tamanho n)
2.   para j=1,2,..., n-1
3.     minimo = j
4.     para k=j+1,j+2,...,n
5.       se v[k] < v[minimo]
6.         minimo = k
7.
8.     troca(v[minimo], v[j])
```

Constante



Algoritmo da Seleção: Complexidade

```
1. função selectionSort(vetor v, tamanho n)
2.   para j=1,2,..., n-1
3.     minimo = j
4.     para k=j+1,j+2,...,n
5.       se v[k] < v[minimo]
6.         minimo = k
7.
8.     troca(v[minimo], v[j])
```

Depende de n



$$T(n) = \sum_{j=1}^{n-1} \left(c + \sum_{k=j+1}^n c' \right)$$

Algoritmo da Seleção: Complexidade

$$\begin{aligned}T(n) &= \sum_{j=1}^{n-1} \left(c + \sum_{k=j+1}^n c' \right) \\&= \sum_{j=1}^{n-1} \left(\sum_{k=j+1}^n c' \right) \\&= \sum_{j=1}^{n-1} (n - j) c' \\&= \sum_{j=1}^{n-1} n - j \\&= \sum_{j=1}^{n-1} n - j \leq \sum_{j=1}^n n - j \\&\leq n - 1 + n - 2 + \dots + n - n \\&\leq n^2 - (1 + 2 + \dots + n) = n^2 - \frac{n^2 + n}{2} \\T(n) &\in O(n^2)\end{aligned}$$

Algoritmo da Seleção: Complexidade

$$T(n) = \sum_{j=1}^{n-1} \left(c + \sum_{k=j+1}^n c' \right)$$

$$= \sum_{j=1}^{n-1} \left(\sum_{k=j+1}^n c' \right)$$

$$= \sum_{j=1}^{n-1} (n - j) c'$$

$$= \sum_{j=1}^{n-1} n - j$$

$$= \sum_{j=1}^{n-1} n - j \leq \sum_{j=1}^n n - j$$

$$\leq n - 1 + n - 2 + \dots + n - n$$

$$\leq n^2 - (1 + 2 + \dots + n) = n^2 - \frac{n^2 + n}{2}$$

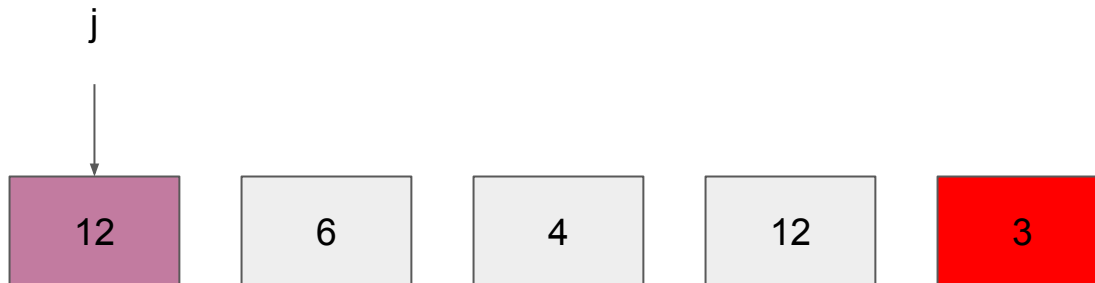
$$T(n) \in O(n^2)$$

Melhor caso?

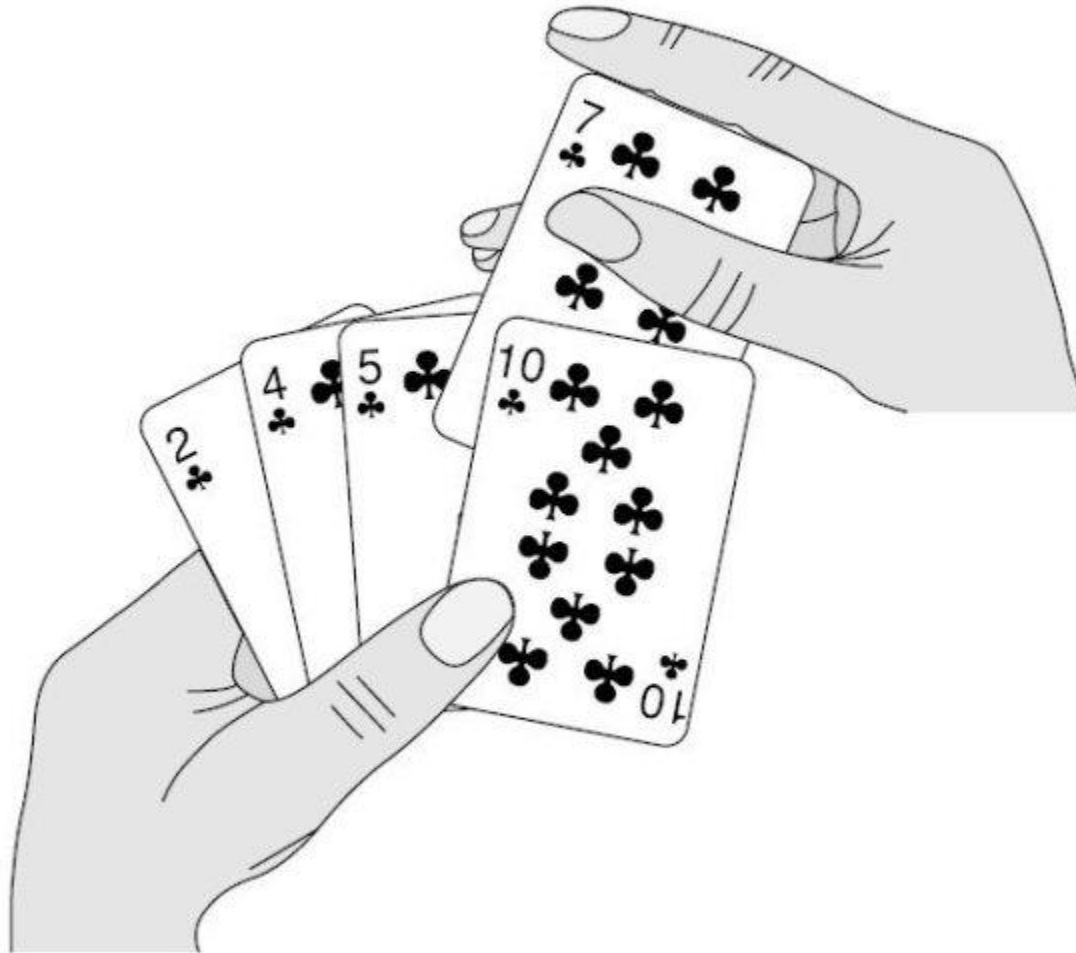
Pior caso?

Algoritmo da Seleção: Estável?

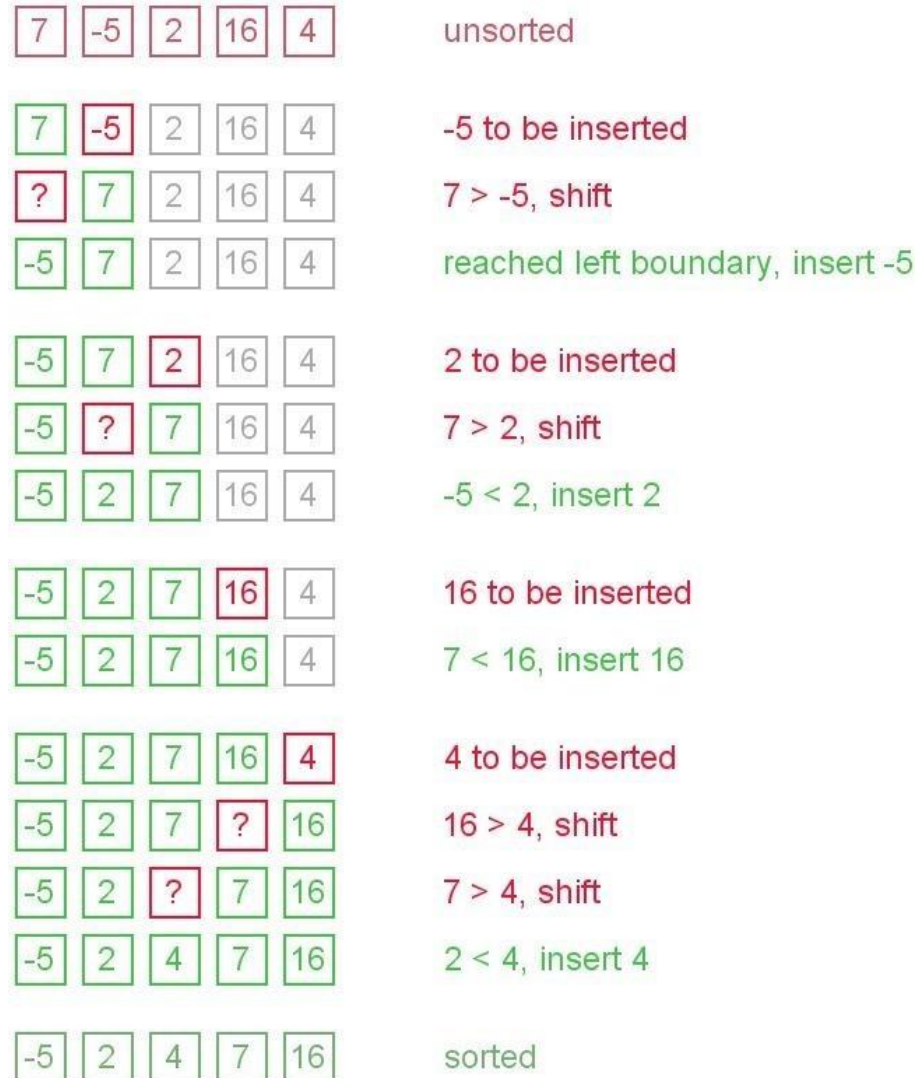
```
1. função selectionSort(vetor v, tamanho n)
2.   para j=1,2,..., n-1
3.     minimo = j
4.     para k=j+1,j+2,...,n
5.       se v[k] < v[minimo]
6.         minimo = k
7.
8.     troca(v[minimo], v[j])
```



Algoritmo da Inserção (*Insertion Sort*)



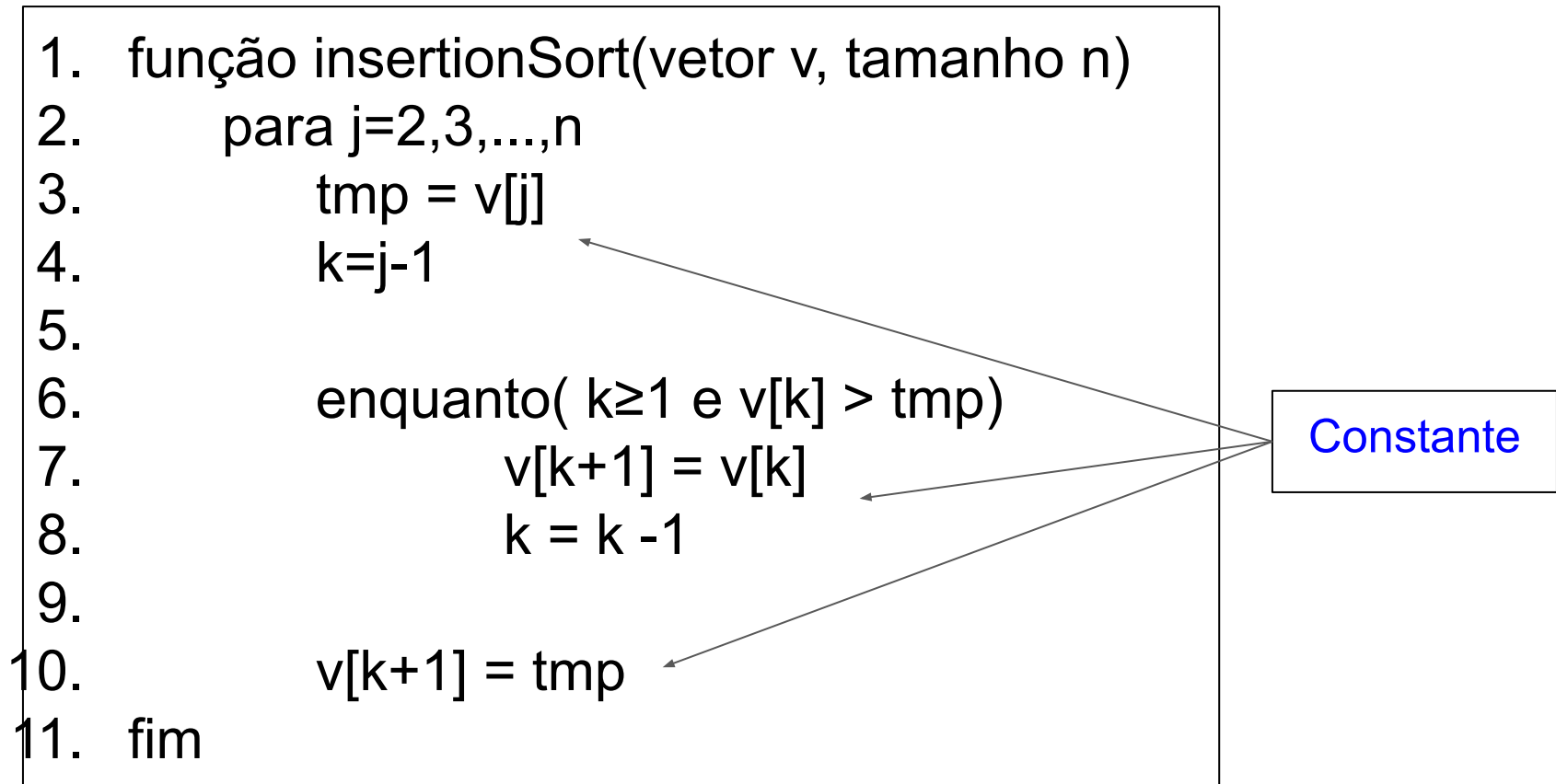
Algoritmo da Inserção: Exemplo



Algoritmo da Inserção

```
1. função insertionSort(vetor v, tamanho n)
2.     para j=2,3,...,n
3.         tmp = v[j]
4.         k=j-1
5.
6.         enquanto( k≥1 e v[k] > tmp)
7.             v[k+1] = v[k]
8.             k = k - 1
9.
10.        v[k+1] = tmp
11. fim
```

Algoritmo da Inserção: Complexidade



Algoritmo da Inserção: Complexidade

```
1. função insertionSort(vetor v, tamanho n)
2.   para j=2,3,...,n
3.     tmp = v[j]
4.     k=j-1
5.
6.     enquanto( k≥1 e v[k] > tmp)
7.       v[k+1] = v[k]
8.       k = k - 1
9.
10.    v[k+1] = tmp
11. fim
```

Depende de n

$$T(n) = \sum_{j=2}^n \left(c + \sum_{k=1}^{j-1} c' \right)$$

Algoritmo da Inserção: Complexidade

$$\begin{aligned}T(n) &= \sum_{j=2}^n \left(c + \sum_{k=1}^{j-1} c' \right) \\&= \sum_{j=2}^n \sum_{k=1}^{j-1} c' \\&= \sum_{j=2}^n (j-1)c' \\&= \sum_{j=2}^n j - 1 \\&\leq \sum_{j=1}^n j - 1 = 0 + 1 + \dots + n = \frac{n^2 + n}{2} \\T(n) &\in O(n^2)\end{aligned}$$

Algoritmo da Inserção: Complexidade

$$T(n) = \sum_{j=2}^n \left(c + \sum_{k=1}^{j-1} c' \right)$$

$$= \sum_{j=2}^n \sum_{k=1}^{j-1} c'$$

$$= \sum_{j=2}^n (j-1)c'$$

$$= \sum_{j=2}^n j - 1$$

$$\leq \sum_{j=1}^n j - 1 = 0 + 1 + \dots + n = \frac{n^2 + n}{2}$$

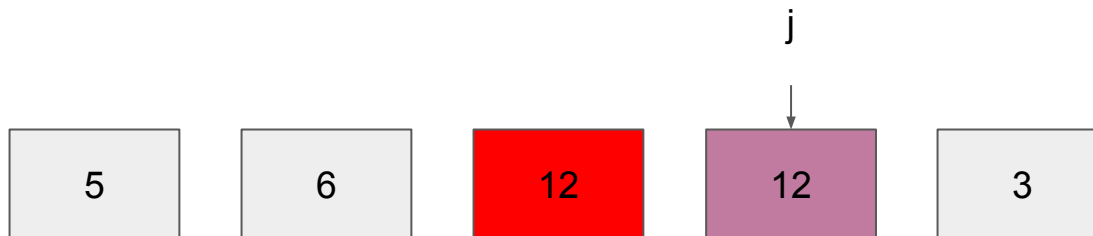
$$T(n) \in O(n^2)$$

Melhor caso?

Pior caso?

Algoritmo da Inserção: Estável?

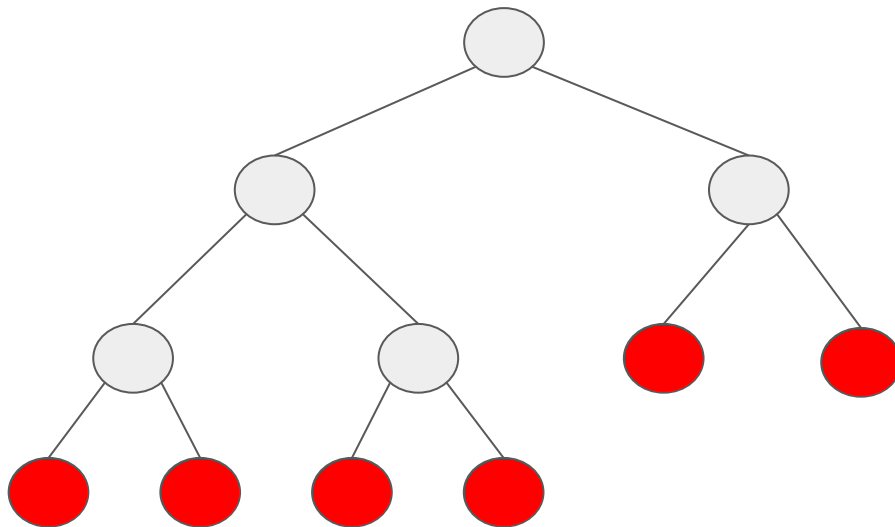
```
1. função insertionSort(vetor v, tamanho n)
2.   para j=2,3,...,n
3.     tmp = v[j]
4.     k=j-1
5.
6.     enquanto( k≥1 e v[k] > v[j])
7.       v[k+1] = v[k]
8.       k = k - 1
9.
10.    v[k+1] = tmp
11. fim
```



Algoritmos Eficientes

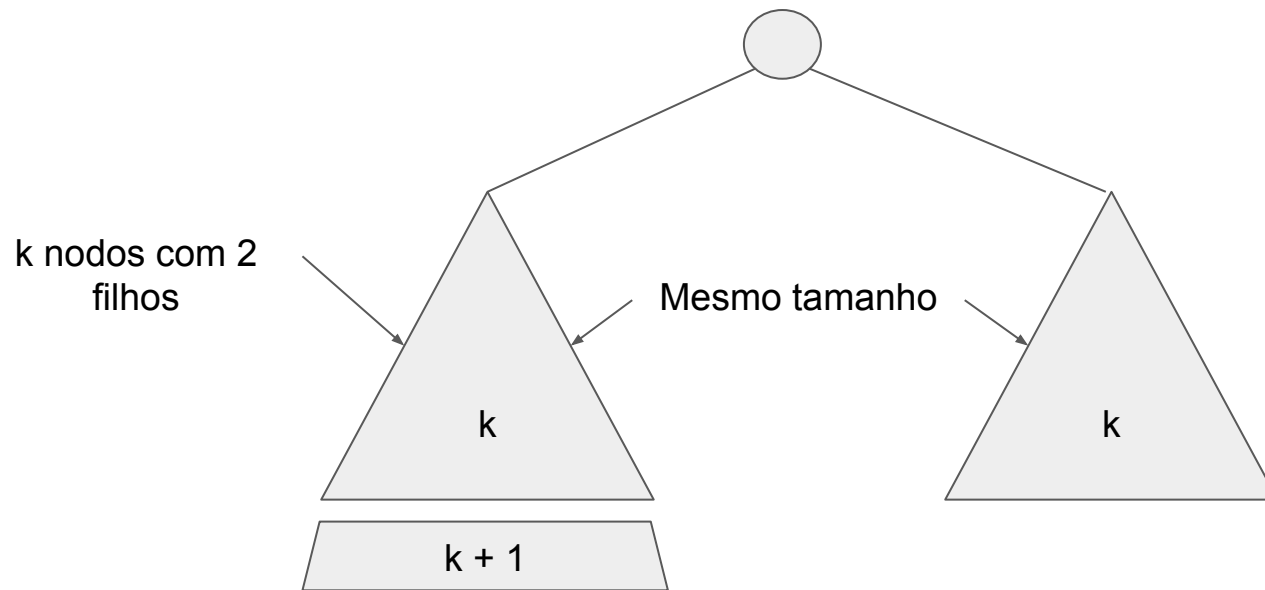
Árvores binárias completas

- Todo nodo tem 2 filhos ou nenhum



#nodos com 2 filhos = k
#nodos com 0 filhos = $k + 1$

Árvores binárias completas



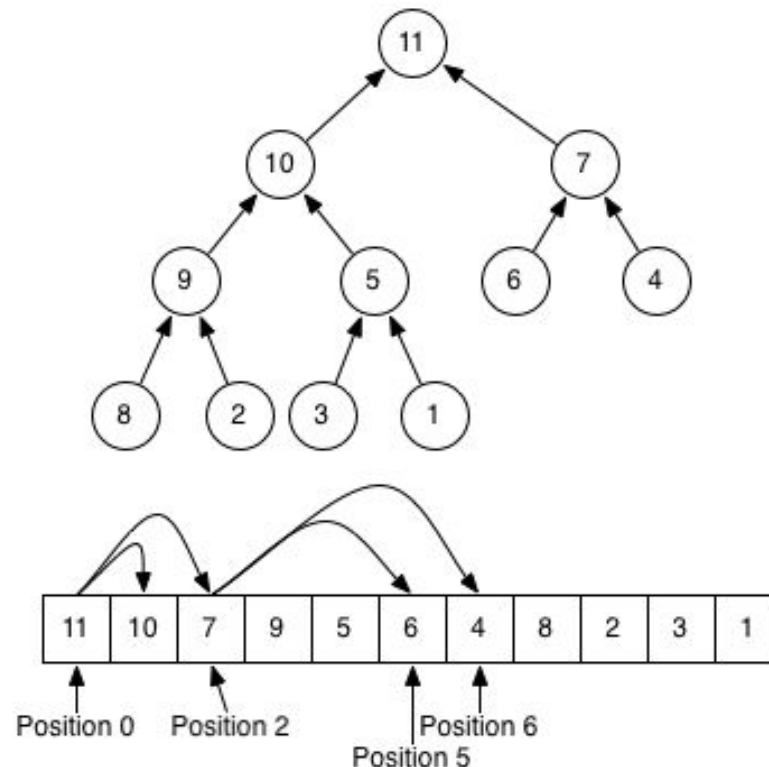
$$\text{Total} = 1 + k + k + 1 + k = 3k+2$$

$$\text{SubárvoreEsq/Total} = (2k+1) / (3k+2) \approx 2/3 \text{ dos nodos para } k \text{ grande}$$

Heap (binária) **máxima**

- Sequência de objetos $\langle o_1, o_2, \dots, o_n \rangle$ que satisfaz a condição

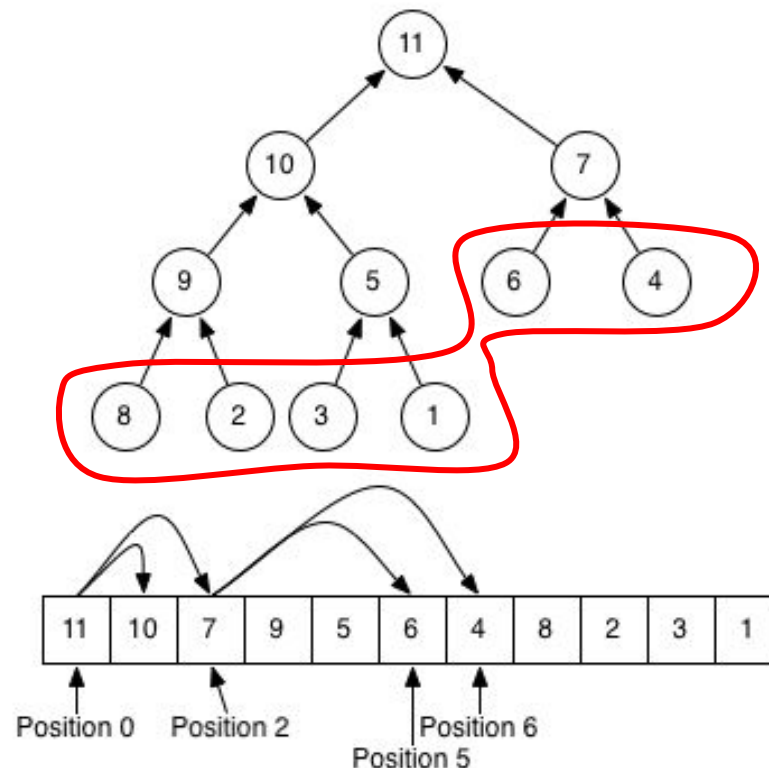
$$o_i \geq o_{2i} \text{ e } o_i \geq o_{2i+1} \text{ para todo } i=1, 2, \dots, n/2$$



Heap máxima

- Sequência de objetos $\langle o_1, o_2, \dots, o_n \rangle$ que satisfaz a condição

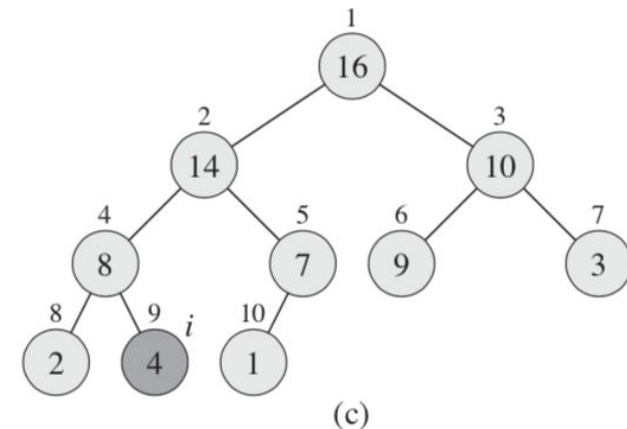
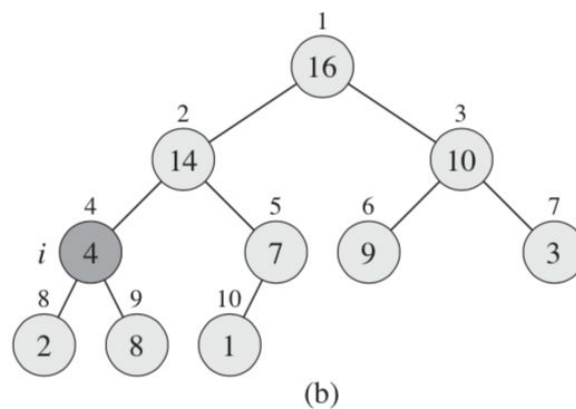
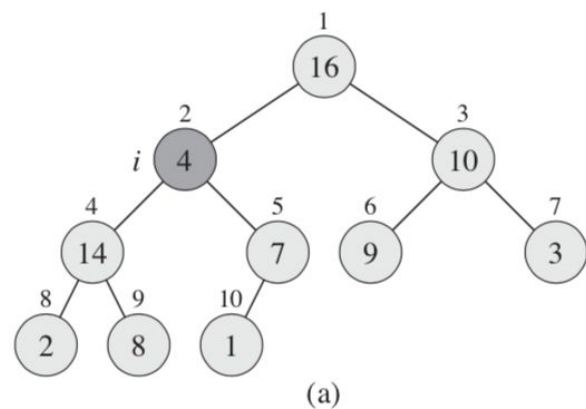
$$o_i \geq o_{2i} \text{ e } o_i \geq o_{2i+1} \text{ para todo } i=1, 2, \dots, n/2$$



Para $i > n/2$ a condição sempre é satisfeita

Função *max-heapify*

- Move o i -ésimo objeto para alguma posição à direita para satisfazer a condição de heap máxima.
- Exemplo: $i=2$ em (a) abaixo



Função *max-heapify*

- Seja $A=\{v,t\}$ uma heap de objetos o_1, o_2, \dots, o_t armazenados no vetor $v=v[1], v[2], \dots, v[n]$ onde $n \geq t > 0$.
- Seja i o i -ésimo objeto o_i da heap

Função *max-heapify*

```
1. função max-heapify(A, i)
2.   esq = i*2
3.   dir = i*2 + 1
4.   maximo = i
5.   se esq ≤ t E A.v[esq] > A.v[i]
6.     maximo = esq
7.   se dir ≤ t E A.v[dir] > A.v[maximo]
8.     maximo = dir
9.   se maximo ≠ i
10.    troca(A.v[i], A.v[maximo])
11.    max-Heapify(A, maximo)
```

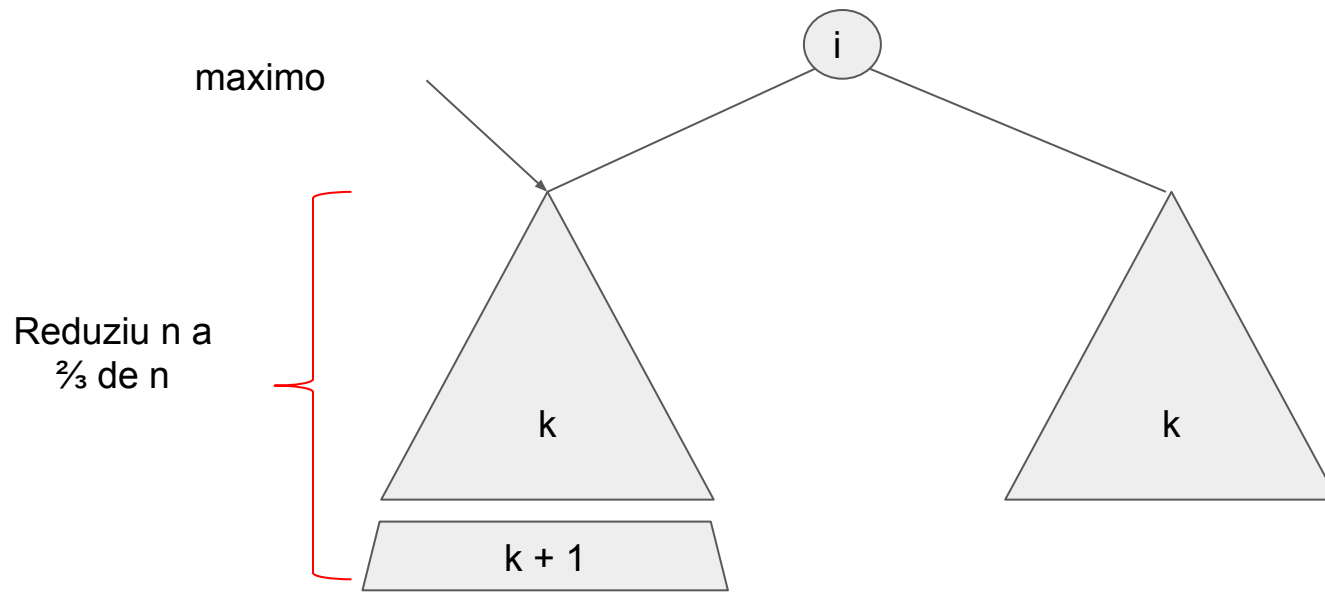
Função *max-heapify*

```
1. função max-heapify(A, i)
2.   esq = i*2
3.   dir = i*2 + 1
4.   maximo = i
5.   se esq ≤ t E A.v[esq] > A.v[i]
6.     maximo = esq
7.   se dir ≤ t E A.v[dir] > A.v[maximo]
8.     maximo = dir
9.   se maximo ≠ i
10.    troca(A.v[i], A.v[maximo])
11.    max-Heapify(A, maximo)
```

Constante: $\Theta(1)$

Depende de n.
Reduziu qto n?

Pior caso: subárvores com alturas diferentes



Função *max-heapify*

```
1. função max-heapify(A, i)
2.   esq = i*2
3.   dir = i*2 + 1
4.   maximo = i
5.   se esq ≤ t E A.v[esq] > A.v[i]
6.     maximo = esq
7.   se dir ≤ t E A.v[dir] > A.v[maximo]
8.     maximo = dir
9.   se maximo ≠ i
10.    troca(A.v[i], A.v[maximo])
11.    max-Heapify(A, maximo)
```

Constante: $\Theta(1)$

Reduz a $\frac{2}{3} n$

$$T(n) = T\left(\frac{2}{3}n\right) + \Theta(1) \in O(\log n)$$

Função *construirHeapMax*

- Organiza vetor v de modo que v seja uma heap máxima

```
1. função construirHeapMax(vetor  $v$ , tamanho  $n$ )  
2.    $A.v = v$   
3.    $A.t = n$   
4.   para  $i = \lfloor n/2 \rfloor, \lfloor n/2 \rfloor - 1, \dots, 1$   
5.       max-heapify( $A, i$ )  
6.   retorne  $A$ 
```

Função *construirHeapMax*

- Organiza vetor v de modo que v seja uma heap máxima

```
1. função construirHeapMax(vetor v, tamanho n)
2.   A.v = v
3.   A.t = n
4.   para i =  $\lfloor n/2 \rfloor, \lfloor n/2 \rfloor - 1, \dots, 1$ 
5.       max-heapify(A,i)
6.   retorne A
```

$O(n \log n)$?

Função *construirHeapMax*: Complexidade

- Organiza vetor v de modo que v seja uma heap máxima

```
1. função construirHeapMax(vetor v, tamanho n)
2.   A.v = v
3.   A.t = n
4.   para i =  $\lfloor n/2 \rfloor, \lfloor n/2 \rfloor - 1, \dots, 1$ 
5.       max-heapify(A,i)
6.   retorne A
```

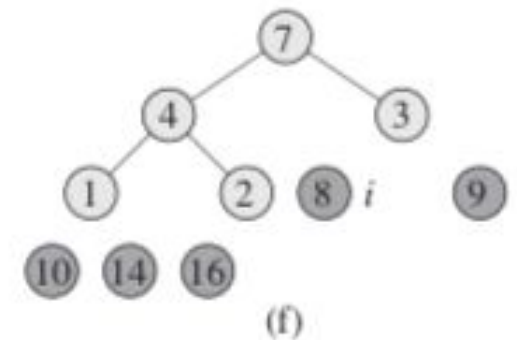
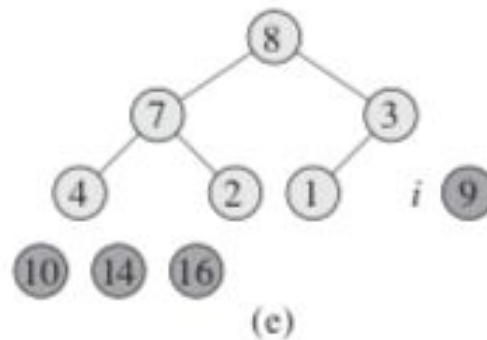
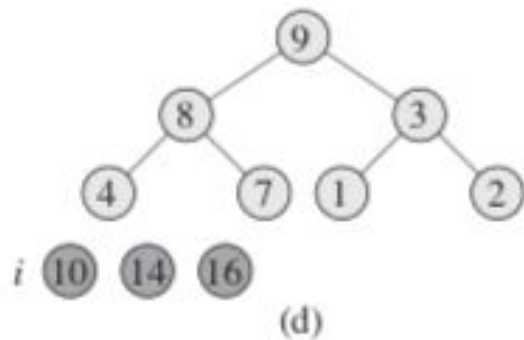
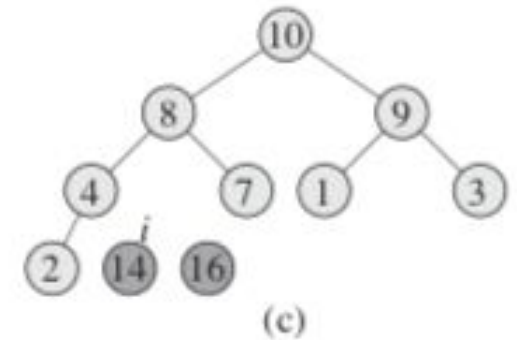
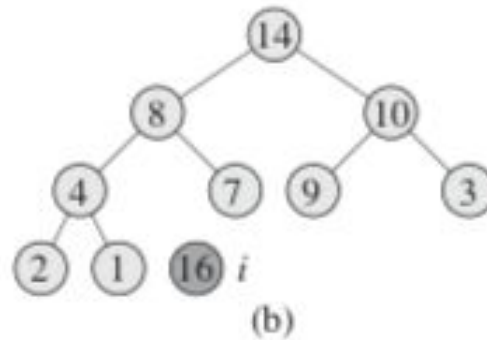
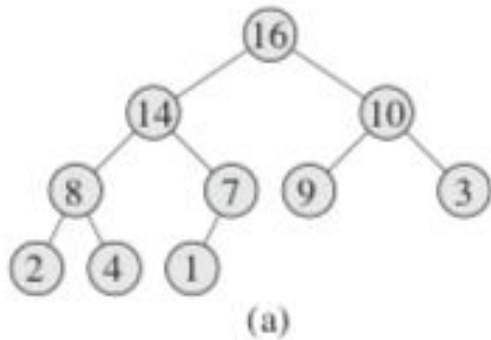
$O(n)$

Ver Cormen, Sec 6.3

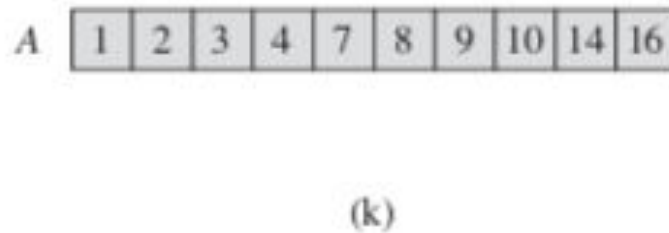
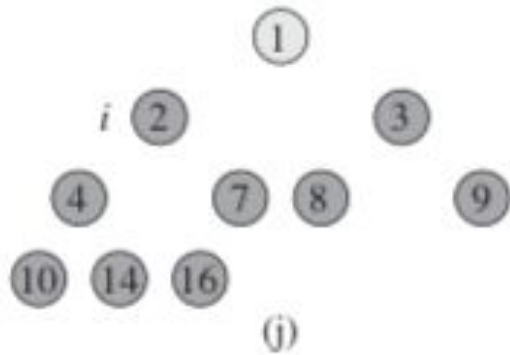
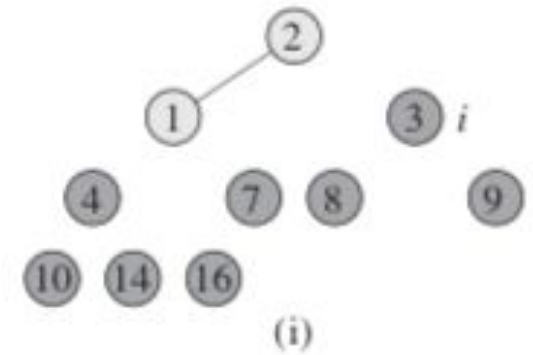
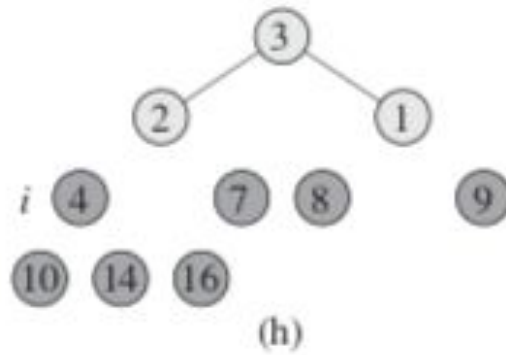
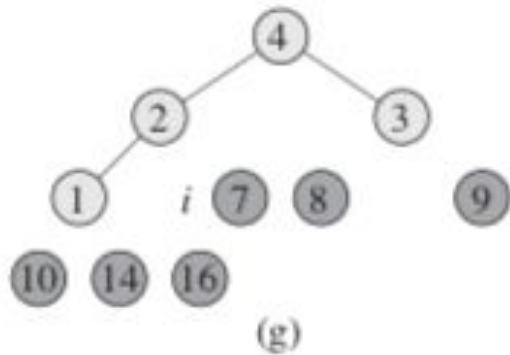
Heap Sort: Ideia

Na i -ésima iteração, troque primeiro objeto da heap de tamanho $n-i$ com $(n-i)$ -ésimo objeto do vetor

Heap Sort: Exemplo (1/2)



Heap Sort: Exemplo (2/2)



Heap Sort: Algoritmo

```
1. função heapSort(vetor v, tamanho n)
2.   A = contruirHeapMáxima(v,n)
3.   para k=n, n-1, ..., 2
4.     trocar(A.v[1], A.v[k])
5.     A.t = A.t - 1
6.     max-heapify(A, 1)
```

Heap Sort: Complexidade

```
1. função heapSort(vetor v, tamanho n)
2.   A = contruirHeapMáxima(v,n)
3.   para k=n, n-1, ..., 2
4.     trocar(A.v[1], A.v[k])
5.     A.t = A.t - 1
6.     max-heapify(A, 1)
```

$O(n)$

Ver Cormen, Sec 6.3

$O(n \log n)$

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