Heap Sort

- Baseia-se na estratégia para projeto de Algoritmo: **Transformar & Conquistar**
- Esta técnica compreende dois estágios:
  a. No estágio de transformação a instância do problema é modificada para ser, por uma razão ou outra, mais fácil para encontrar uma solução
  b. No segundo estágio a instância "transformada" é resolvida

Transformar e Conquistar

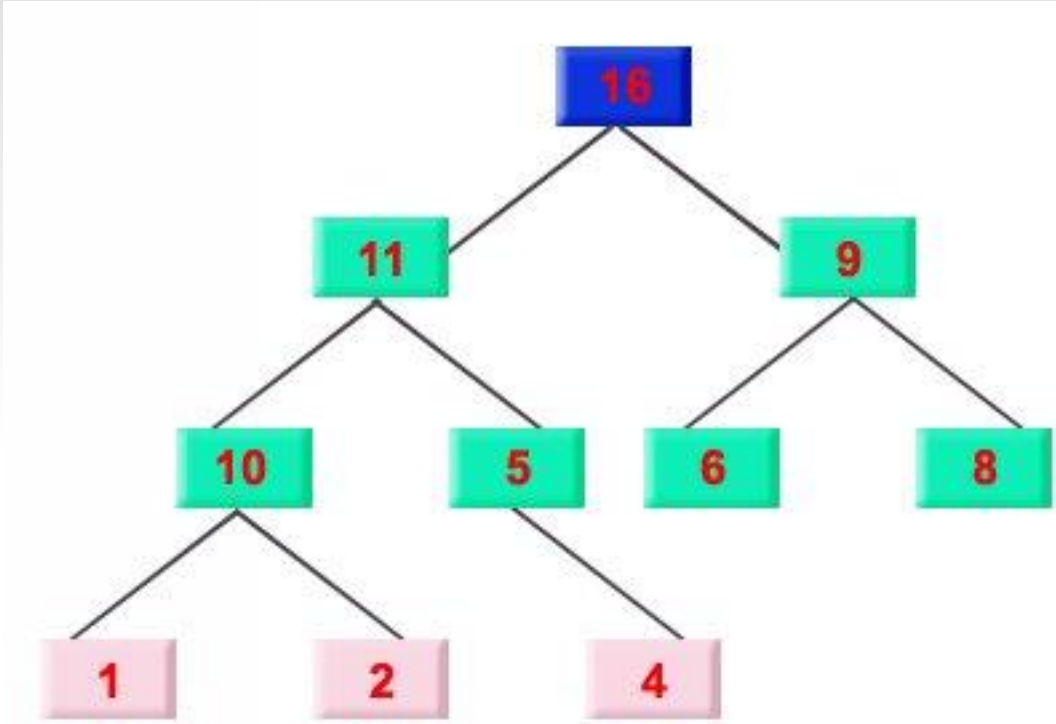
- Existem três variações principais desta idéia, que se diferem no que se transforma uma dada instância:
  - Transformação para uma instância mais simples ou mais conveniente do mesmo problema (**simplificação da instância**)
    - Ex: Eliminação Gaussiana, pré-ordenação, etc
  - Transformação para uma representação diferente da mesma instância (**mudança da representação**)
    - Ex: Árvores de busca balanceadas, ***Heap e HeapSort***, etc
  - Transformação para uma instância de um problema diferente para o qual já existe um algoritmo eficaz (**redução do problema**)
    - Reduções a problemas de grafos, Programação linear, etc

Heaps e HeapSort

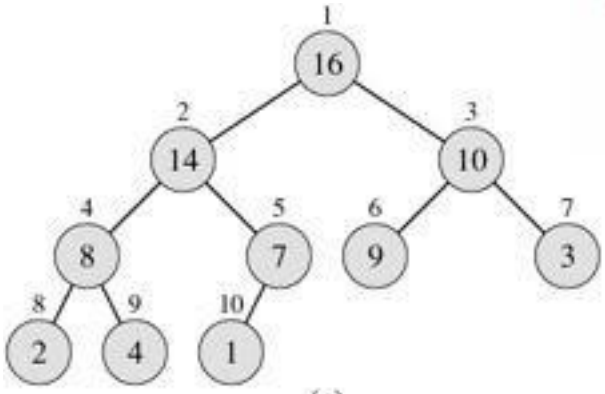
- ***Heap*** é uma estrutura de dados parcialmente ordenada que é adequada para a implementação de filas de prioridade
  - **Fila de prioridade:** conjunto de itens com uma característica ordenável chamada **prioridade de um item**, a partir das seguintes operações:
    - Encontrar um elemento com a prioridade mais elevada
    - Apagar um item com a mais alta prioridade
    - Adicionar um novo item ao conjunto
    - Usados em: S.O.s, Sistemas de gerência de memória, etc

Heap
Definição

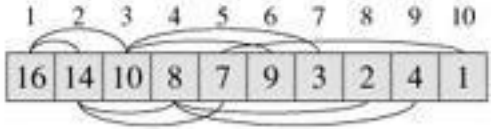
- ***Heap*** é uma estrutura de dados em forma de **árvore binária balanceada** onde o **nó pai é sempre maior que o nó filho**



Heap with 10 items



(a)



(b)

Heap
Propriedades

1. Existe exatamente uma árvore binária essencialmente completa com n nós. Sua altura é igual a $lg_2 n$
2. A raiz do *heap* sempre contém o maior ou menor elemento
   - Heap Máximo: $A[pai[i]] \geq A[i]$
   - Heap Mínimo: $A[pai[i]] \leq A[i]$
3. Um nó de um *heap* com todos seus descendentes é também um *heap*
4. Um heap pode ser implementado como um arranjo, gravando seus elementos de maneira cima–baixo (*top–down*) e esquerda–direita (*left–to–right*)

Heap
Propriedades (II)

- No ***Heap*** para o arranjo *H[1..n]* tem-se que:
  - As chaves pais estarão nas *n/2* primeiras posições do arranjo. As chaves folhas ocuparão as últimas *n/2* posições
    - $H[i] \geq max\ \{H[2i], H[2i+1]\}$ *para* $i = 1,..., n/2$
  - O filho de uma chave na posição pai "*i*" ($1 \leq i \leq n/2$) estará nas posições *2i* e *2i+1*
    - *Esquerda(i): return 2i*
    - *Direita(i): return 2i+1*
  - Consequentemente, o pai de uma chave na posição "*i*" ($2 \leq i \leq n$) estará na posição *i/2*
    - *Pai(i): return i/2*

Heap
Representação por *array*

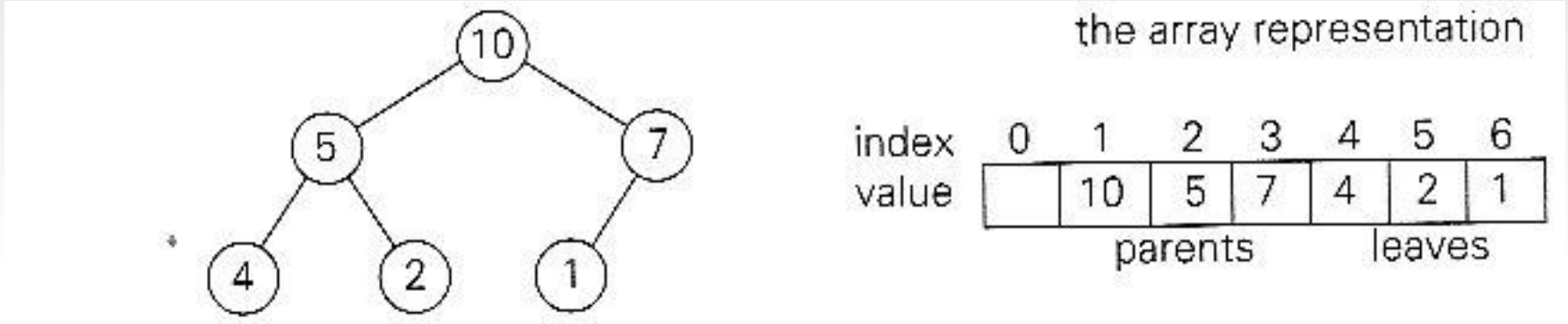
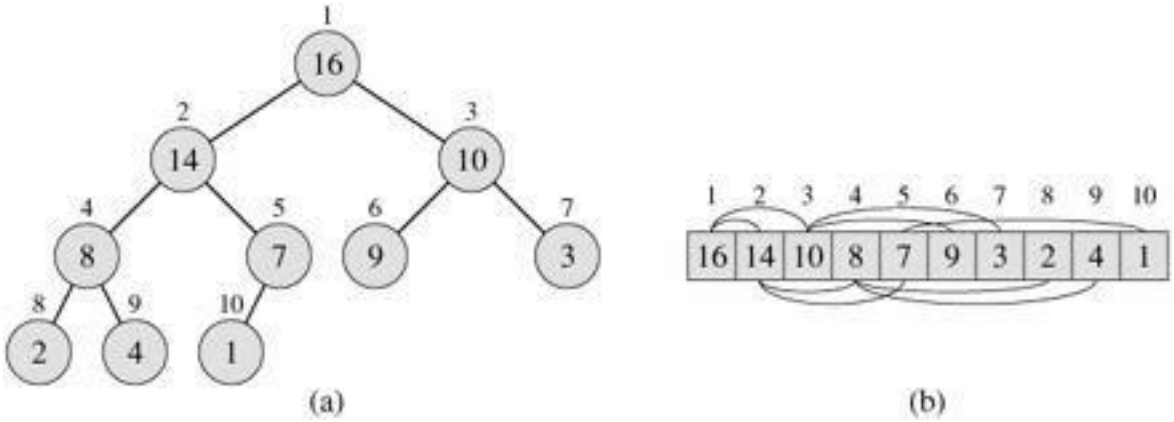


FIGURE 6.10 Heap and its array representation



Heap
Exemplo de construção

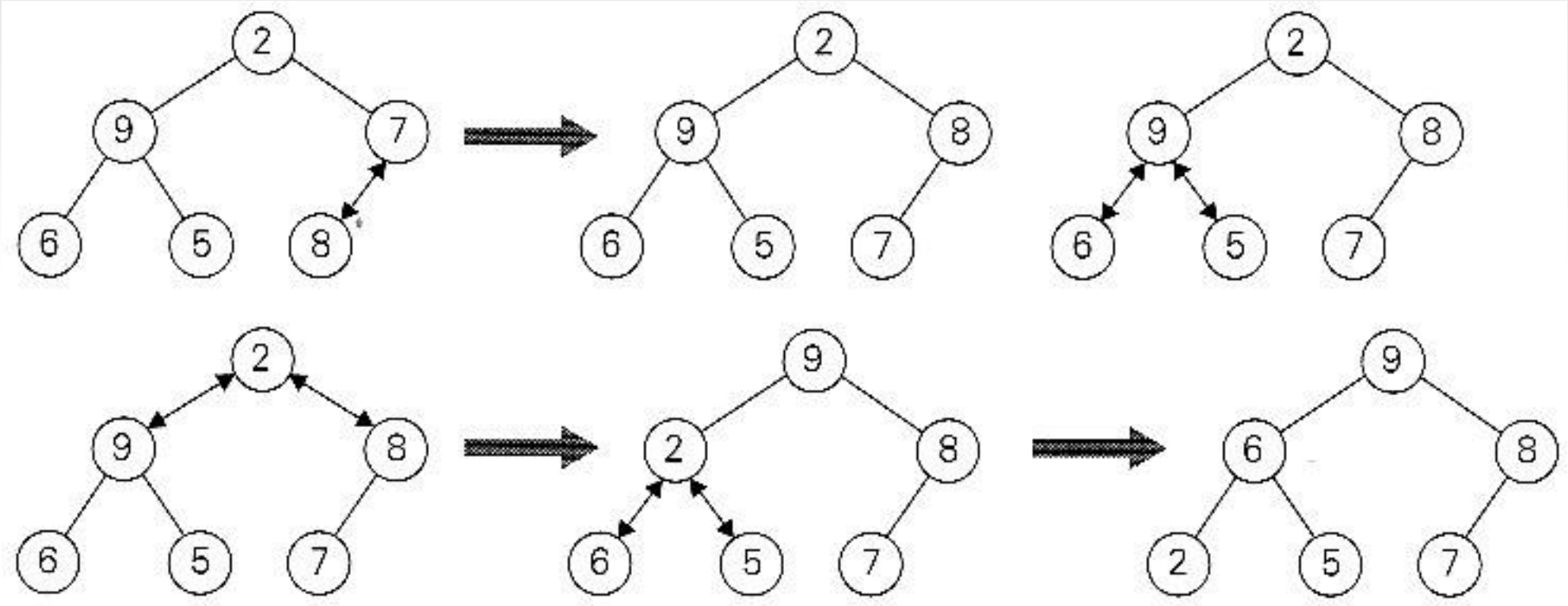


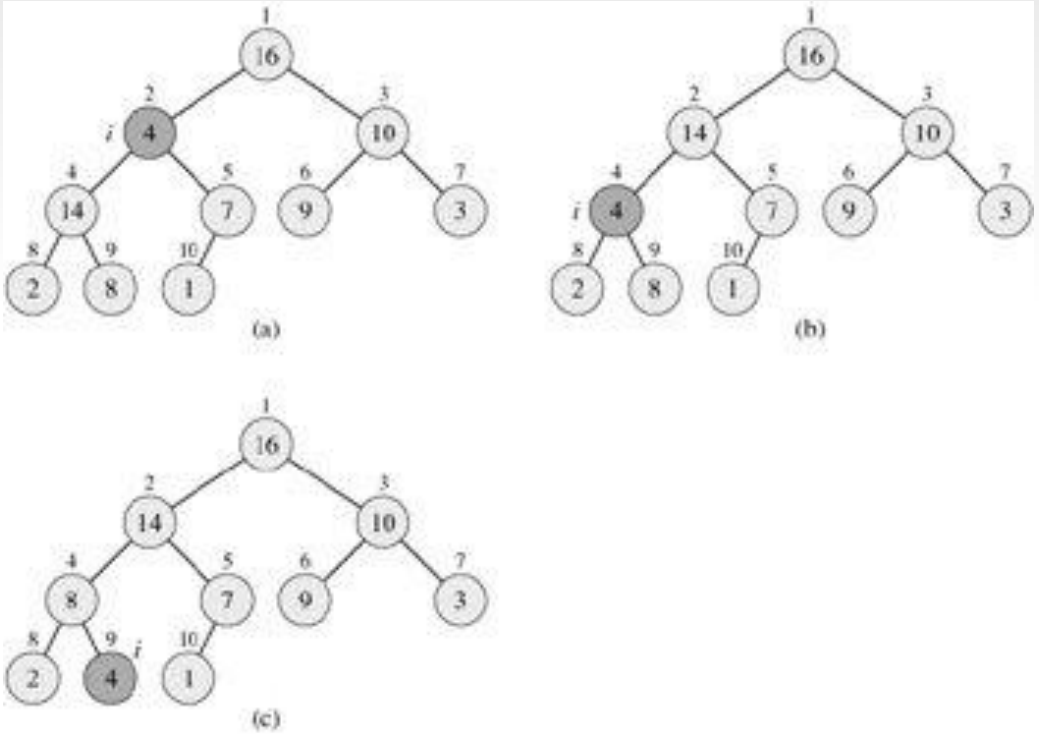
FIGURE 6.11 Bottom-up construction of a heap for the list 2, 9, 7, 6, 5, 8

Manutenção da propriedade de um *Heap*

```
   MAX-HEAPIFY(A, i)
1) l ← left(i)   //2*i
2) r ← right(i) //2*i+1
3) if ((l≤lenght[A]) and (A[l]>A[i]))
4)       max ← l
5) else max ← i
6) if ((r≤lenght[A]) and (A[r]>A[max]))
7)       max ← r
8) if (max≠i) {
9)       swap(A[i],A[max])
10)      MAX-HEAPIFY(A, max) }
```

Exemplo de uso do MAX-HEAPIFY

- (a) *A[2]* viola propriedade
- (b) troca de *A[2]* por *A[4]*, mas *A[4]* viola a propriedade
- (c) troca de *A[4]* por *A[9]*
  - Chamadas recursivas a *MAX-HEAPIFY*



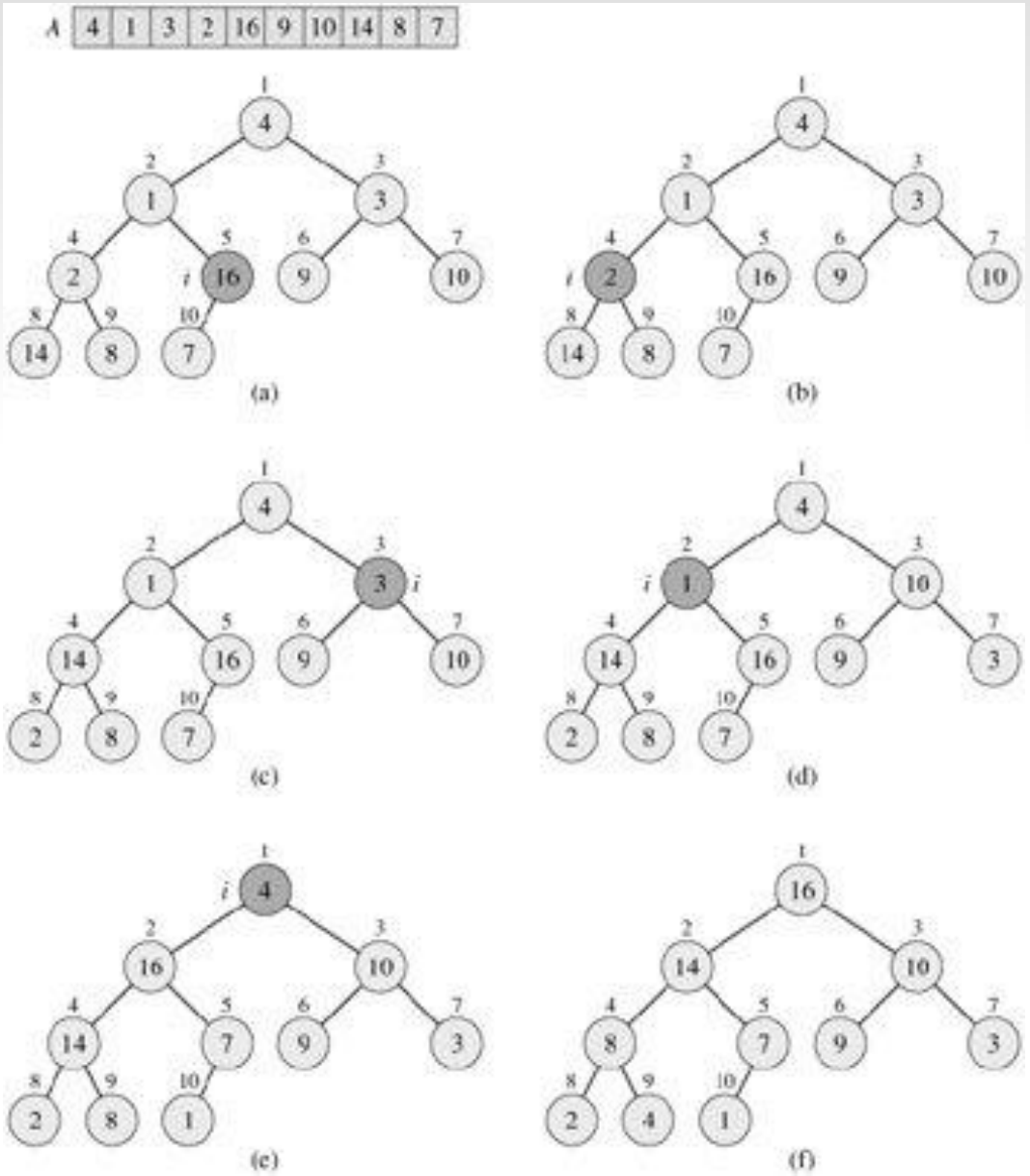
Construção de um *Heap*

```
BUILD-MAX-HEAP(A)
1) n←lenght[A]
2) for i←n/2 downto 1
3)     MAX-HEAPIFY(A, i)
```

- Usa-se o procedimento *MAX-HEAPIFY* de baixo para cima
  - Converte *A[1..n]* em um *heap* máximo
  - Cada nó *i+1, i+2, ..., n* é a raiz de um *heap* máximo

Exemplo de uso de *BUILD-MAX-HEAP*

- (a) arranjo original
  - Laço em *i* se refere ao nó 5 inicialmente.
- (b) Resultado
  - Laço *i* no nó 4
- (c)-(e) iterações subsequentes
- (f) resultado final



HeapSort
Algoritmo

- Aplicar a operação de apagamento da raiz *n–1* vezes sobre o *heap* resultante, ou seja:
  1. Remover raiz – trocar com a última folha (mais a direita)
  2. Arrumar o *heap* (excluindo a última folha)
  3. Repetir o passo 1 até que o heap contenha somente um nó

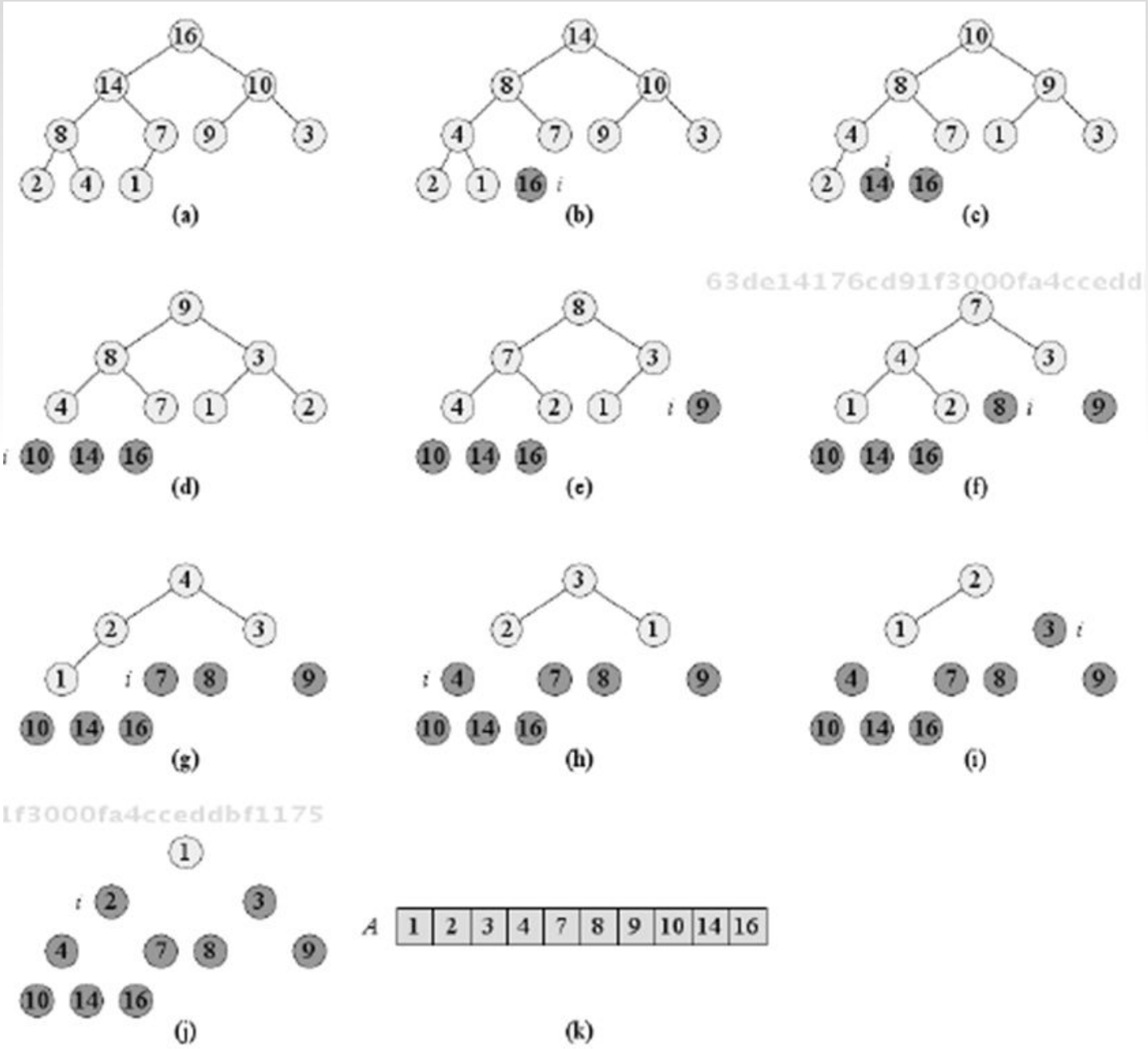
HEAP-EXTRACT-MAX
Algoritmo

```
HEAP-EXTRACT-MAX(A)
1)    if (lenght[A]<1)
2)       error(”Underflow”)
3)    max ← A[1]
4)    last ← lenght[A]
5)    A[1] ← A[last]
6)    lenght[A]--
7)    max-HEAPFY(A,1)
return max
```

Algoritmo *HeapSort*

```
   HEAPSORT(A)
1) BUILD-MAX-HEAP(A)
2)    n ← lenght[A]
3)    int V[n] //Vetor saida
4)    While(n>0) do
5)          V[n] ← HEAP-EXTRACT-MAX(A)
6)          n--;
```

HeapSort – Exemplo



Análise do *Heap Sort*

- *BUILD-MAX-HEAP* tem complexidade $\Theta(n)$
- *MAX-HEAPIFY*
  - É executado $n - 1$ vezes
  - *MAX-HEAPIFY* tem complexidade $\Theta(\log n)$ no pior caso
  - Executa no máximo uma vez a cada nível
  - A cada nível, executa um número constante de operações
- Então, o *Heapsort*, no pior caso, tem complexidade de:

  $(n - 1)\Theta(\log n) + \Theta(n) = \Theta(n \log n)$

Resumo dos métodos

- Para um vetor de tamanho *n*:

	Melhor caso	Pior caso	Caso Médio
BubleSort	$O(n^2)$	• $\Omega(n^2)$	• $\Theta(n^2)$
SelectionSort	$O(n^2)$	• $\Omega(n^2)$	• $\Theta(n^2)$
MergeSort	$O(n \lg n)$	• $\Omega(n \lg n)$	• $\Theta(n \lg n)$
QuickSort	• $O(n \lg n)$	• $\Omega(n^2)$	• $\Theta(n \lg n)$

- *QuickSort* possui notável eficiência na média
  - Tempos relativos aos fatores ocultos (constante de custo por chamada) na implementação bastante curtos
    - Apesar de ter $\Omega(n^2)$ no pior caso e ser recursivo
- Pode-se combinar o uso dos métodos
- Todos métodos de ordenação até aqui fazem comparações