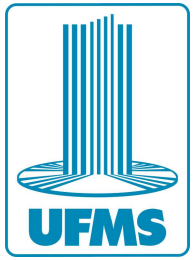
Implementação Algorítmica

Heap Sort

Carlos H. A. Higa

Faculdade de Computação
Universidade Federal de Mato Grosso do Sul



Motivação

- Como o MERGE-SORT, o HEAPSORT tem tempo de execução $O(n \lg n)$;
- Como o INSERTION-SORT, o HEAPSORT não necessita de um vetor auxiliar;
- O HEAPSORT introduz outra técnica de projeto de algoritmos: usa uma estrutura de dados chamada *heap* para gerenciar informações;
- *heap*s são usados em muitas outras aplicações, tais como listas de prioridades (escalonamento de processos em um computador, simulação de uma lista de eventos, etc), coleta de lixo, entre outras.

Heaps

Definição

Um **max heap** é uma coleção de elementos identificados por suas prioridades, armazenadas em um vetor numérico A satisfazendo a seguinte propriedade:

$$A[\lfloor i/2 \rfloor] \geq A[i] \,, \tag{1}$$

para todo $i > 1$.

- Um vetor A que representa um *heap* é um objeto com dois atributos: $A.length$ e $A.heap\text{-}size$, com $0 \leq A.heap\text{-}size \leq A.length$;
- A propriedade (1) é chamada **propriedade max-heap**;
- Um vetor A com a propriedade (1) é chamado um **max-heap**.

Heaps

- É mais interessante ver um max-heap como uma árvore binária;
- Isso nos permite verificar a propriedade (1) mais facilmente;
- O conteúdo de um vértice da árvore é maior ou igual aos conteúdos dos vértices que são seus filhos.

Heaps

- Olhando para um *heap* como uma árvore, definimos a **altura** de um vértice no *heap* como o número de arestas no caminho mais longo deste vértice até uma folha;
- A altura do *heap* é a altura de sua raiz;
- Como um *heap* de n elementos é baseado em uma árvore binária completa, sua altura é $\Theta(\lg n)$;
- As operações básicas sobre *heap*s têm tempo máximo proporcional à altura da árvore e assim têm tempo $O(\lg n)$.

Heaps

PARENT(i)

1 **return** $\lfloor i/2 \rfloor$

LEFT(i)

1 **return** $2i$

RIGHT(i)

1 **return** $2i + 1$

Heaps

- A propriedade max-heap (1) pode ser reescrita:

$$A[\textsc{Parent}(i)] \geq A[i] \,, \tag{2}$$

  para todo nó i, menos a raiz.
- Em um min-heap teríamos:

$$A[\textsc{Parent}(i)] \leq A[i] \,. \tag{3}$$

Heap Sort

Veremos os seguintes procedimentos:

- MAX-HEAPIFY, que é $O(\lg n)$, que mantém a propriedade max-heap;
- BUILD-MAX-HEAP, procedimento de **tempo linear**, que produz um max-heap a partir de um vetor não ordenado;
- HEAPSORT, que executa em tempo $O(n \lg n)$, e ordena um vetor.

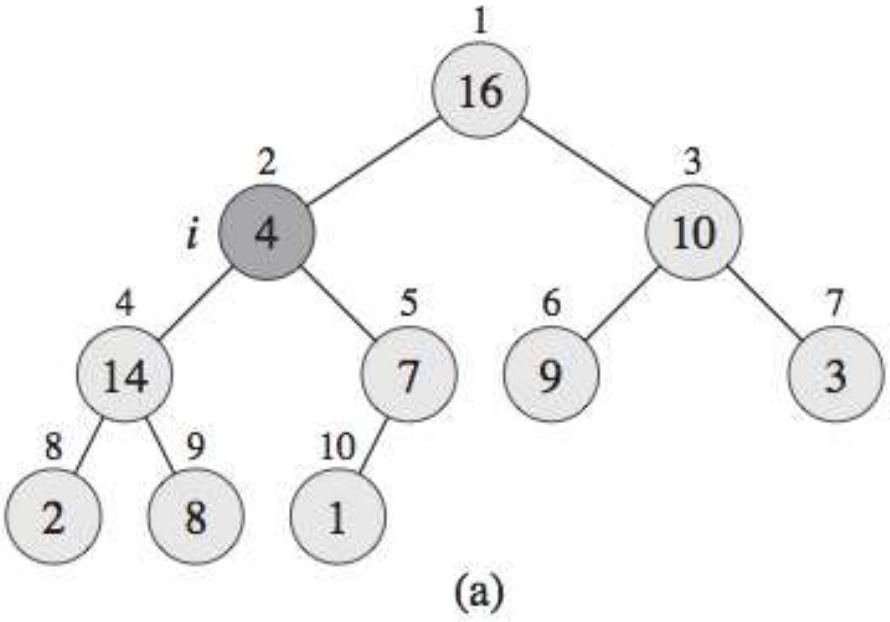
Heap Sort

MAX-HEAPIFY(A, i)

```
1  l = LEFT(i)
2  r = RIGHT(i)
3  if l ≤ A.heap-size and A[l] > A[i]
4        largest = l
5  else largest = i
6  if r ≤ A.heap-size and A[r] > A[largest]
7        largest = r
8  if largest ≠ i
9        exchange A[i] with A[largest]
10       MAX-HEAPIFY(A, largest)
```

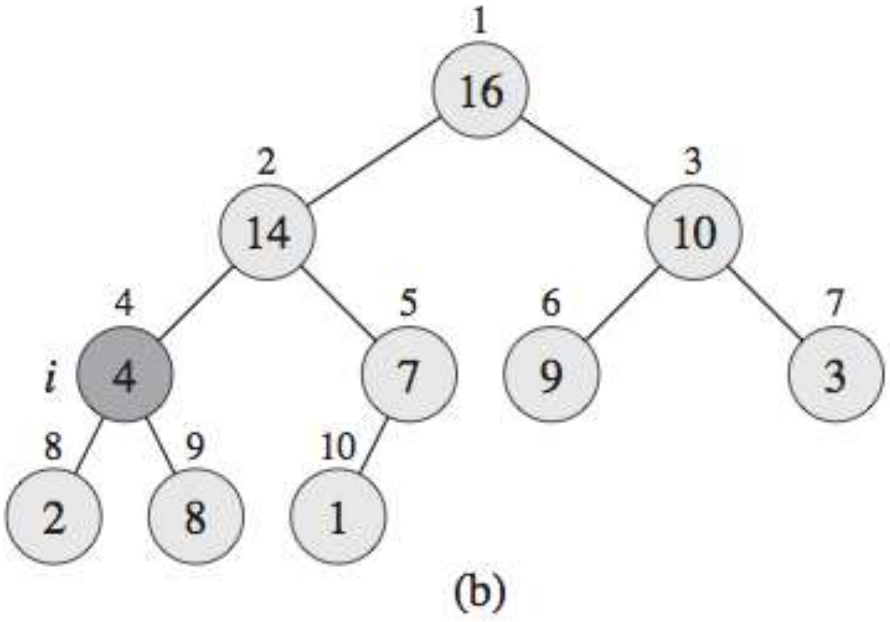
Heap Sort

Chamada de MAX-HEAPIFY$(A, 2)$:



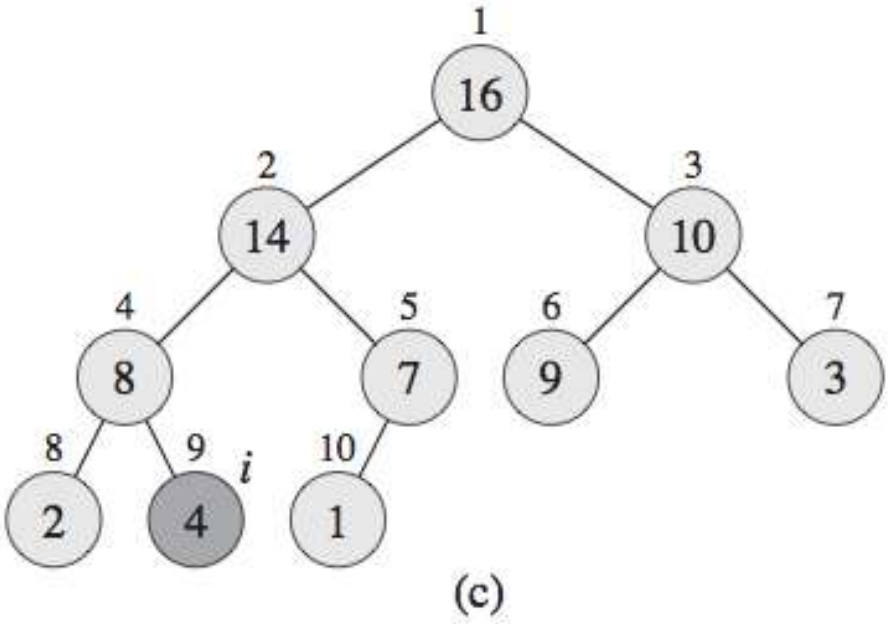
Heap Sort

Chamada de MAX-HEAPIFY$(A, 2)$:



Heap Sort

Chamada de MAX-HEAPIFY$(A, 2)$:



Heap Sort

Tempo de execução do algoritmo MAX-HEAPIFY

- O tempo de execução do algoritmo MAX-HEAPIFY sobre uma subárvore de tamanho n com raiz em um vértice i é dado da seguinte forma:
  - As linhas $1 - 8$ determinam o maior valor dentre aqueles armazenados em $A[i], A[\text{LEFT}(i)]$ e $A[\text{RIGHT}(i)]$ e o tempo gasto é $\Theta(1)$;
  - Mais o tempo da chamada recursiva ao algoritmo MAX-HEAPIFY sobre um dos filhos do vértice i, se ocorrer;
  - Uma subárvore filha do vértice i tem no máximo $2n/3$ vértices.

Heap Sort

Tempo de execução do algoritmo MAX-HEAPIFY

- Assim, o tempo de execução de pior caso do algoritmo MAX-HEAPIFY é dado pela recorrência

$$T(n) \leq T(2n/3) + \Theta(1) \, .$$

- Usando o Teorema Mestre, temos que $a = 1, b = 3/2$ e $f(n) = c$, para alguma constante $c > 0$. Então, $n^{\log_{3/2} 1} = n^0 = 1$ e assim $f(n) = 1 = \Theta(1) = \Theta(n^{\log_{3/2} 1}) = \Theta(n^{\log_b a})$. Portanto, usando o caso 2, temos que $T(n) = \Theta(\lg n)$.

Heap Sort

Problema

Dado um vetor A de números inteiros com $n > 0$ elementos, transformar A em um max-heap.

Heap Sort

Problema

Dado um vetor A de números inteiros com $n > 0$ elementos, transformar A em um max-heap.

BUILD-MAX-HEAP(A)

```
1  A.heap-size = A.length
2  for i = ⌊A.length/2⌋ downto 1
3      MAX-HEAPIFY(A, i)
```

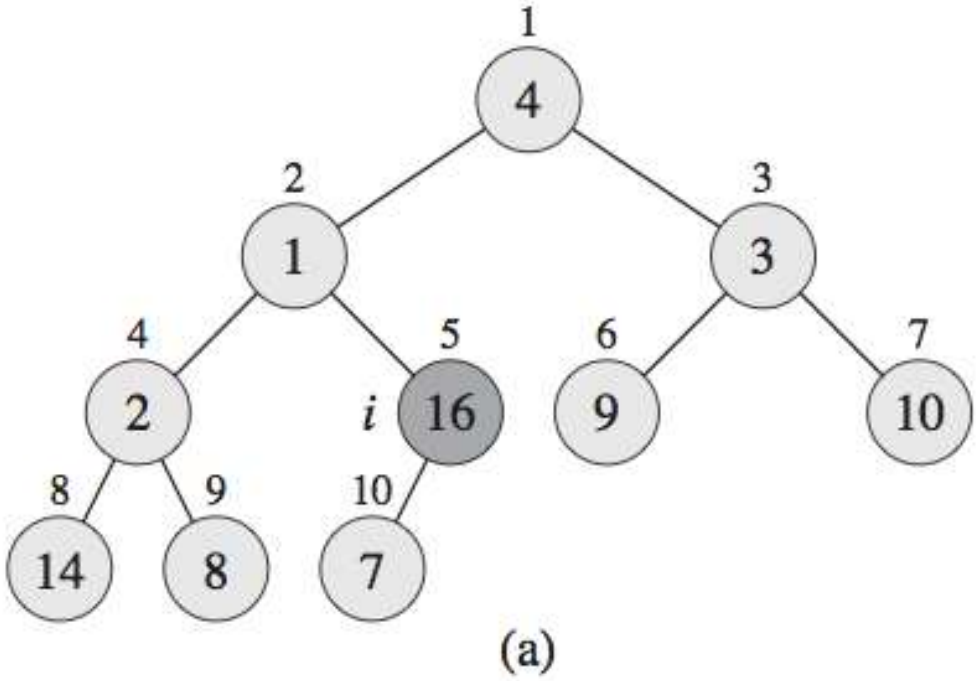
Heap Sort

BUILD-MAX-HEAP(A):

A	4	1	3	2	16	9	10	14	8	7

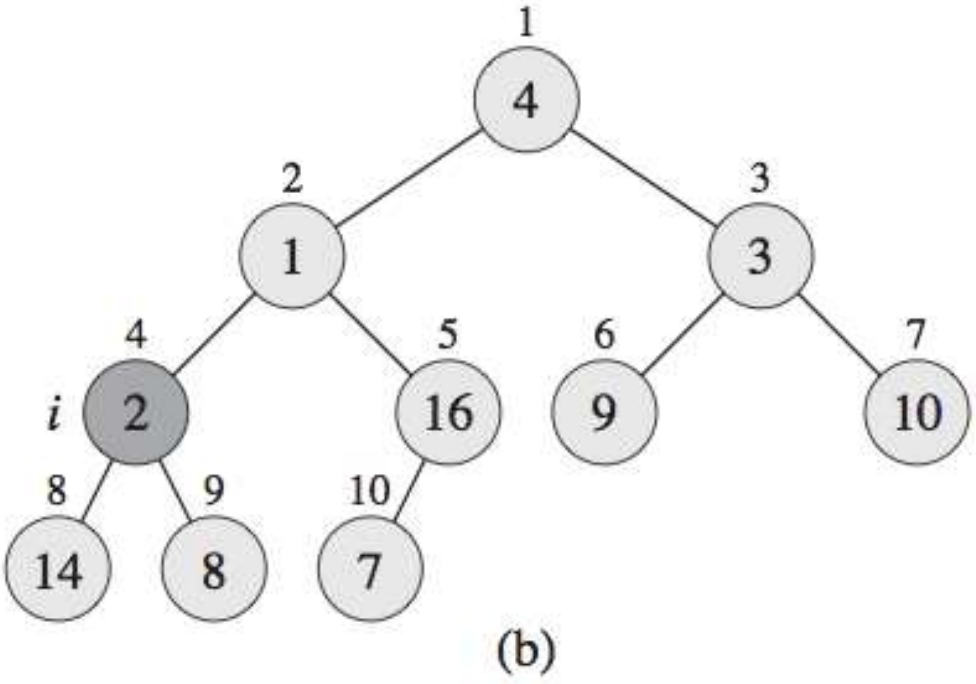
Heap Sort

BUILD-MAX-HEAP(A):



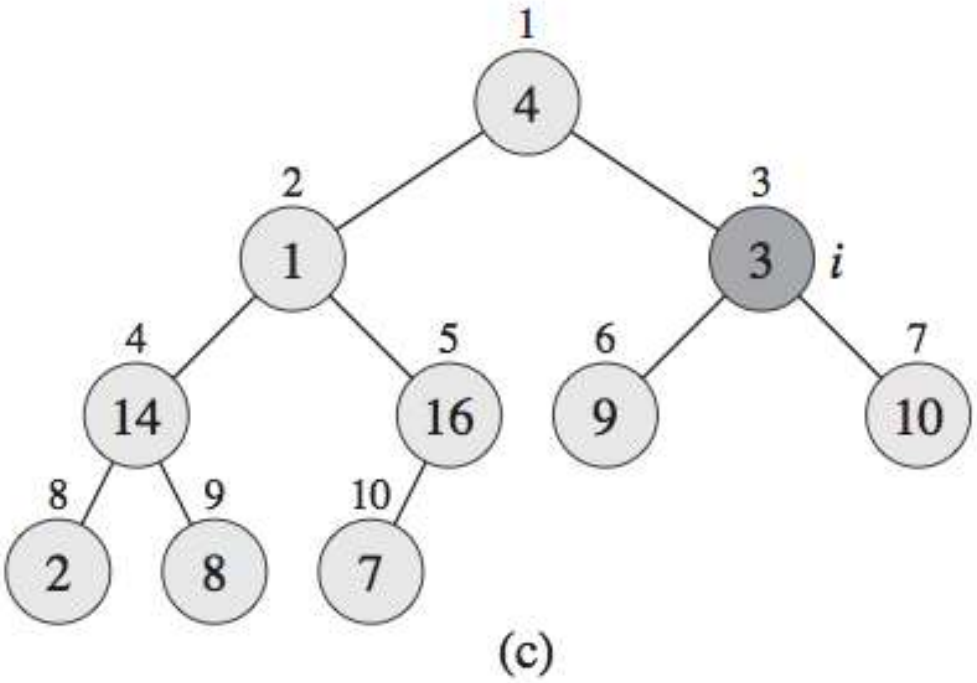
Heap Sort

BUILD-MAX-HEAP(A):



Heap Sort

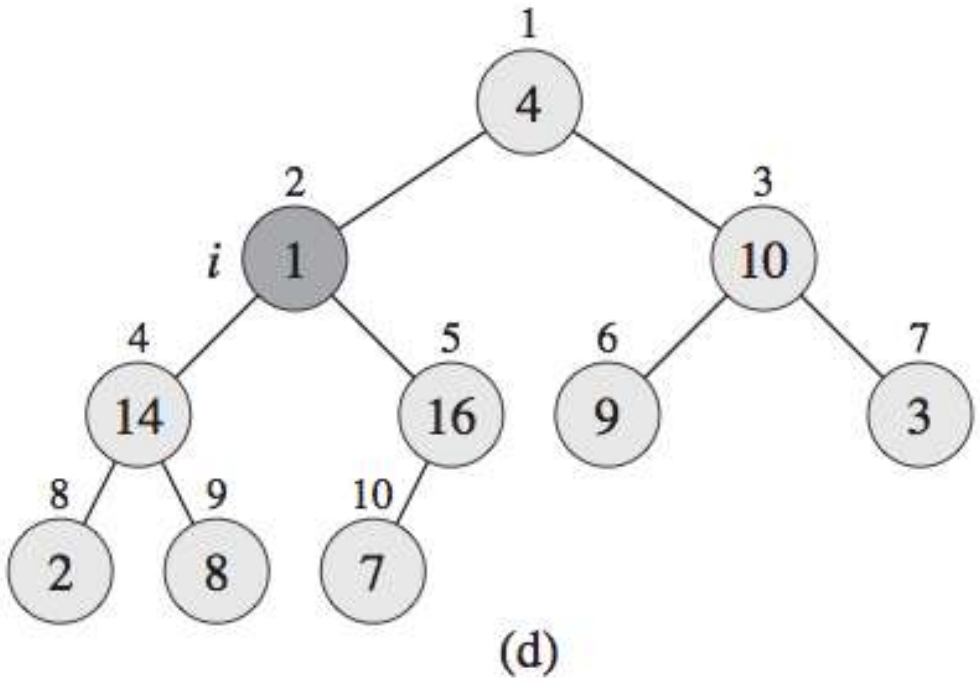
BUILD-MAX-HEAP(A):



(c)

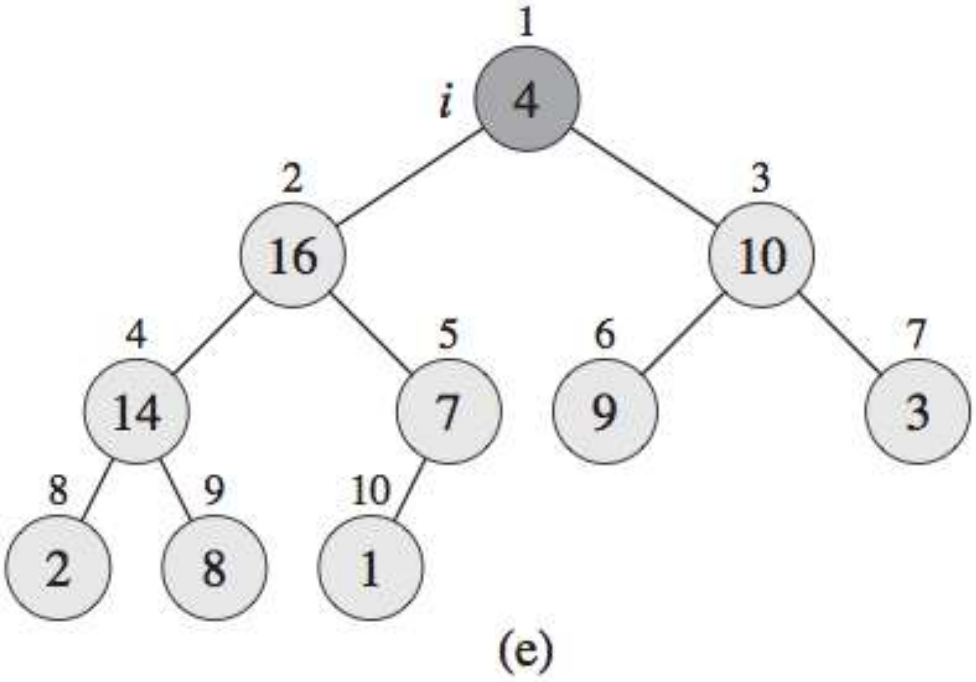
Heap Sort

BUILD-MAX-HEAP(A):



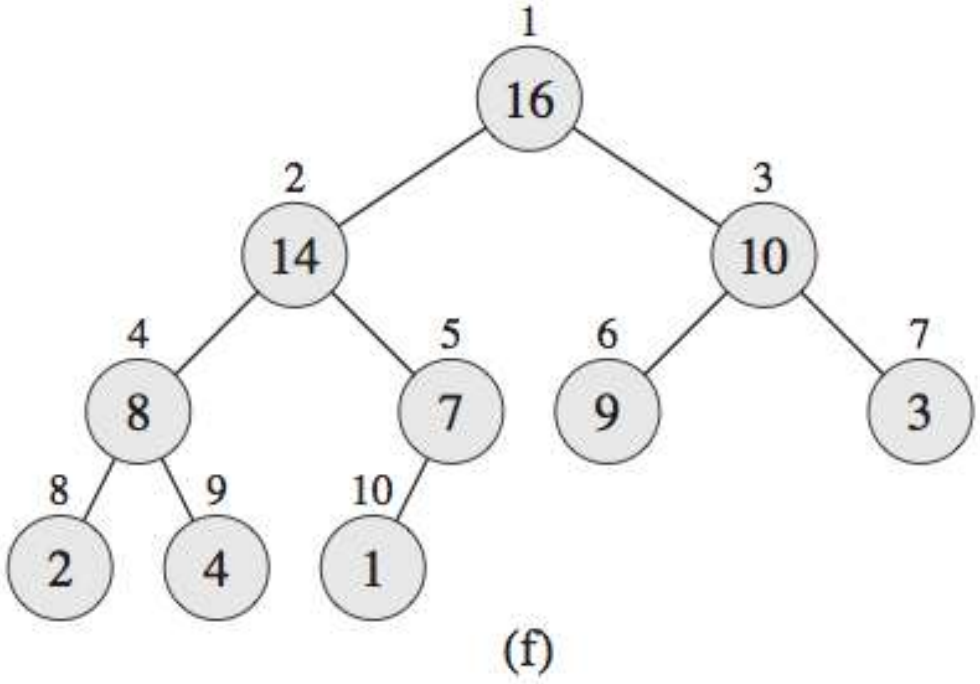
Heap Sort

BUILD-MAX-HEAP(A):



Heap Sort

BUILD-MAX-HEAP(A):



(f)

Heap Sort

Correção do algoritmo BUILD-MAX-HEAP

Invariante: no início de cada iteração da estrutura de repetição das linhas $2-3$, cada vértice $i+1, i+2, \ldots, n$ é raiz de um max-heap.

Devemos mostrar que este invariante é verdadeiro antes da primeira iteração da estrutura de repetição, que cada iteração da estrutura de repetição mantém o invariante e que o invariante fornece uma propriedade útil para mostrar a correção quando a estrutura de repetição termina.

Heap Sort

- Inicialização: Antes da primeira iteração da estrutura de repetição, $i = \lfloor n/2 \rfloor$. Cada vértice $\lfloor n/2 \rfloor + 1, \lfloor n/2 \rfloor + 2, \ldots, n$ é uma folha e, assim, é raiz de uma max-heap trivial.
- Manutenção: Pelo invariante, os filhos de um vértice i são raizes de max-heaps. Então, a chamada MAX-HEAPIFY(A, i) faz com que o vértice i seja a raiz de um max-heap. Além disso, a chamada a MAX-HEAPIFY preserva a propriedade que os vértices $i + 1, i + 2, \ldots, n$ são raizes de max-heaps. No final da iteração, com o decremento de i, o invariante é reestabelecido para a próxima iteração.
- Término: No final, $i = 0$. Pelo invariante, cada vértice $1, 2, \ldots, n$ é raiz de um max-heap. Em particular, o vértice 1 é a raiz de um max-heap.

Heap Sort

Limitante superior para o tempo de execução:

- Primeira análise: o algoritmo MAX-HEAPIFY tem tempo de execução $O(\lg n)$ e é chamado $O(n)$ vezes pelo algoritmo BUILD-MAX-HEAP; portanto, o tempo de execução do algoritmo BUILD-MAX-HEAP é $O(n \lg n)$

Limite mais justo

- Observe que o tempo de MAX-HEAPIFY depende da altura do nó onde é aplicado, e que a maioria dos nós possui uma altura baixa;
- Um heap com n elementos possui altura $\lfloor \lg n \rfloor$ e no máximo $\lceil n/2^{h+1} \rceil$ nós com altura h;
- Assim, o tempo gasto por MAX-HEAPIFY quando chamado em um nó de altura h é $O(h)$.

Heap Sort

Custo total de BUILD-MAX-HEAP:

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

Temos que:

$$\sum_{h=0}^{\lfloor \lg n \rfloor} \frac{h}{2^h} \leq \sum_{h=0}^{\infty} \frac{h}{2^h} = \frac{1/2}{(1-1/2)^2} = 2 .$$

Assim, podemos limitar o tempo de execução de BUILD-MAX-HEAP por:

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

Heap Sort

- Um max-heap pode ser naturalmente usado para descrever um algoritmo de ordenação eficiente;
- O algoritmo constrói um max-heap do vetor de entrada e, como um maior elemento do vetor está armazenado na raiz do max-heap, troca esse elemento com o último elemento do vetor, decrementa o tamanho do max-heap e desce o elemento da raiz;

Heap Sort

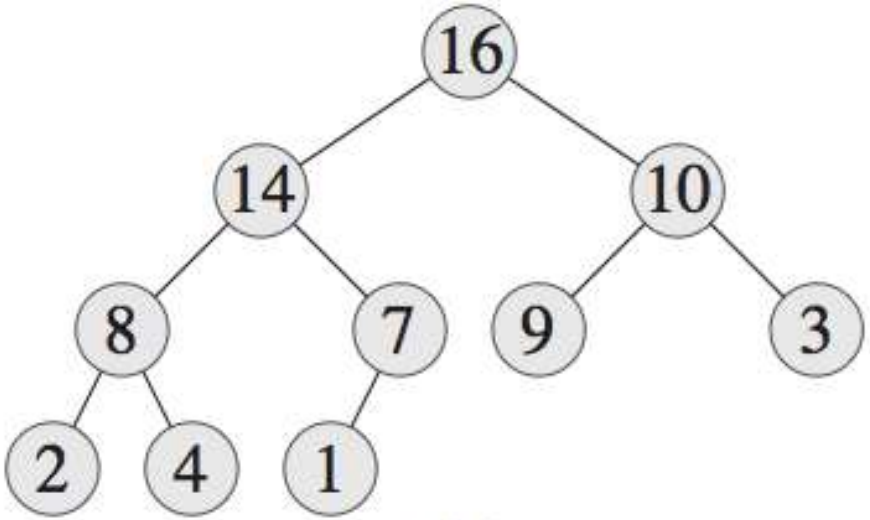
HEAPSORT(A)

```
1  BUILD-MAX-HEAP(A)
2  for i = A.length downto 2
3      exchange A[1] with A[i]
4      A.heap-size = A.heap-size − 1
5      MAX-HEAPIFY(A, 1)
```

HEAPSORT leva tempo $O(n \lg n)$, uma vez que a chamada de BUILD-MAX-HEAP leva tempo $O(n)$ e cada uma das $n - 1$ chamadas a MAX-HEAPIFY leva tempo $O(\lg n)$.

Heap Sort

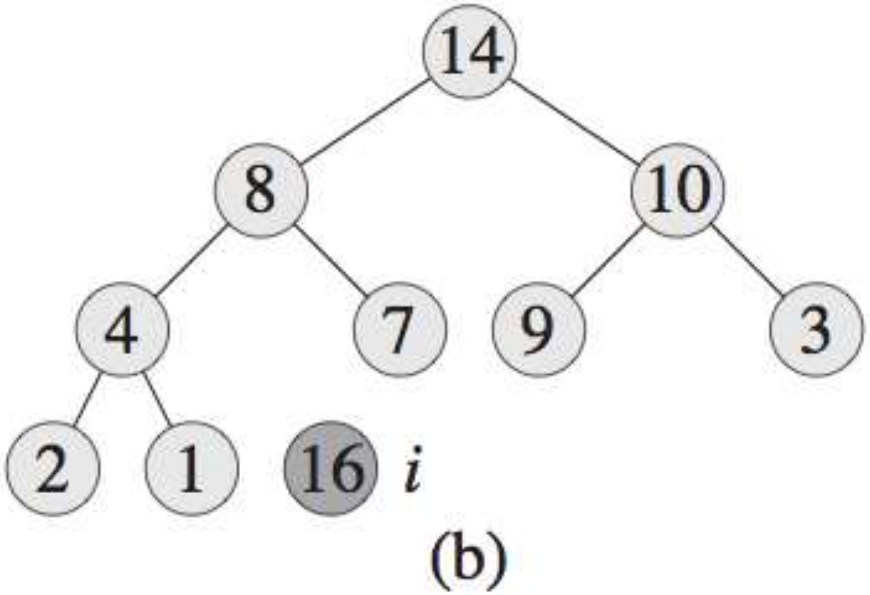
HEAPSORT(A):



(a)

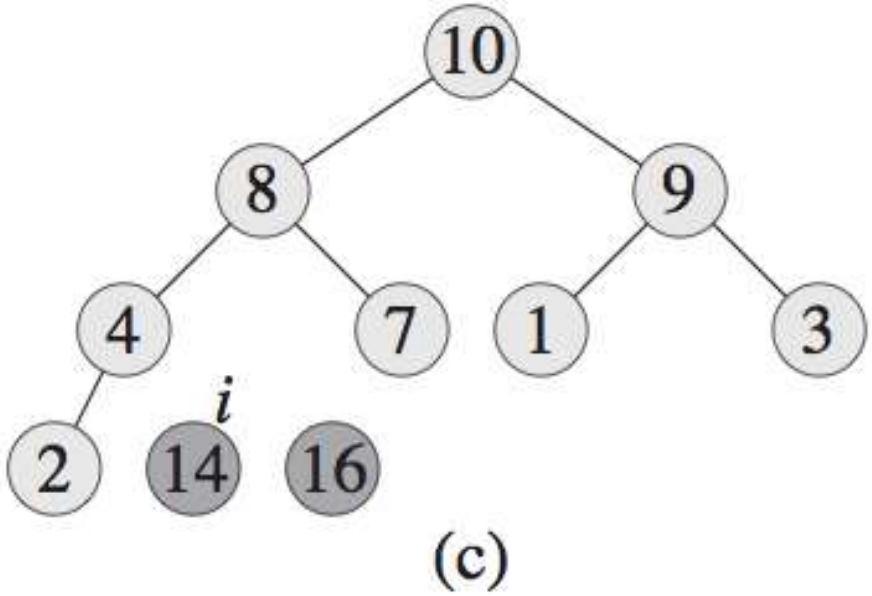
Heap Sort

HEAPSORT(A):



Heap Sort

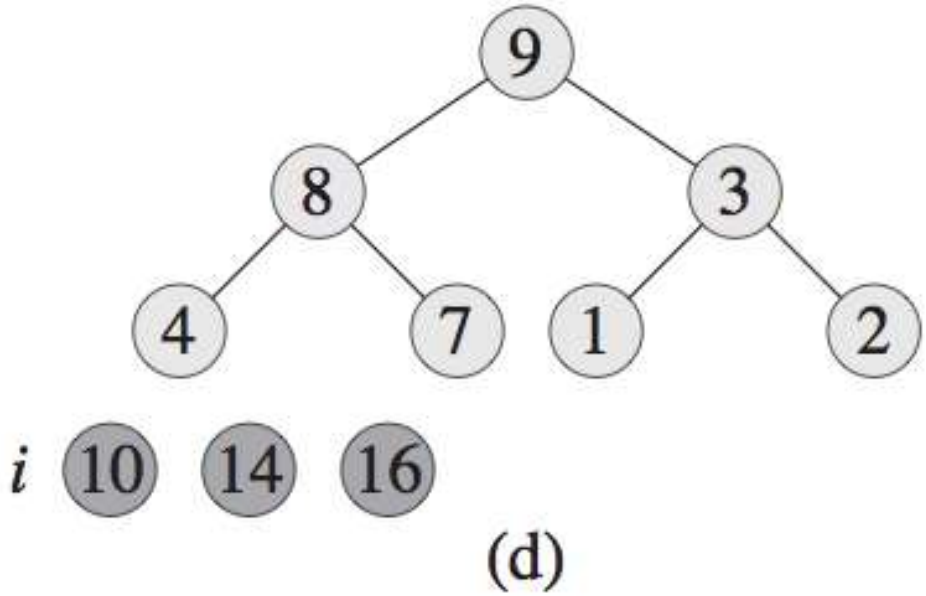
HEAPSORT(A):



(c)

Heap Sort

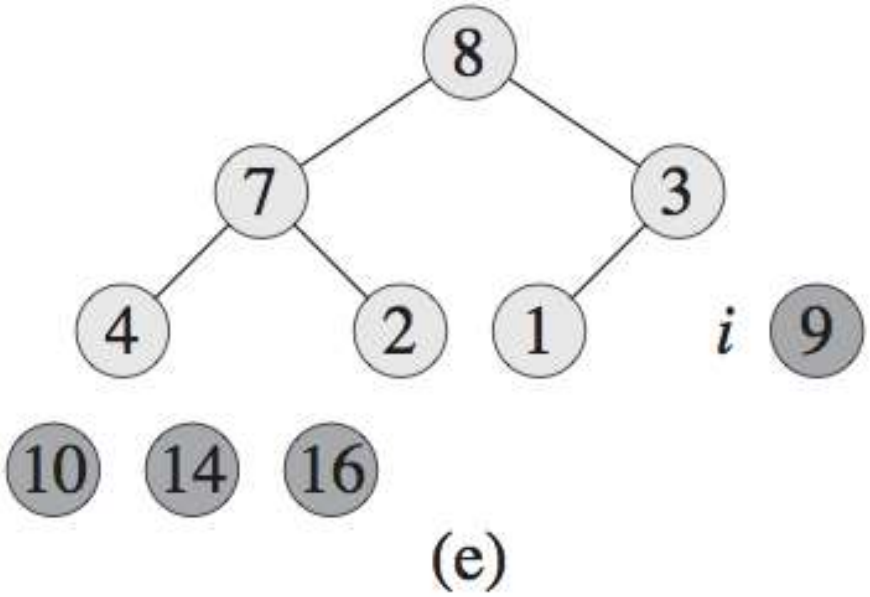
HEAPSORT(A):



(d)

Heap Sort

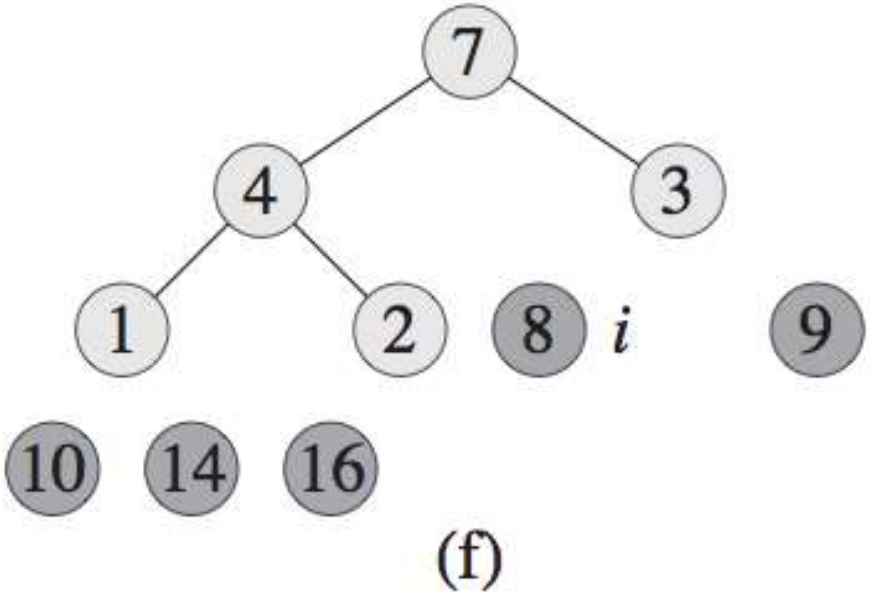
HEAPSORT(A):



(e)

Heap Sort

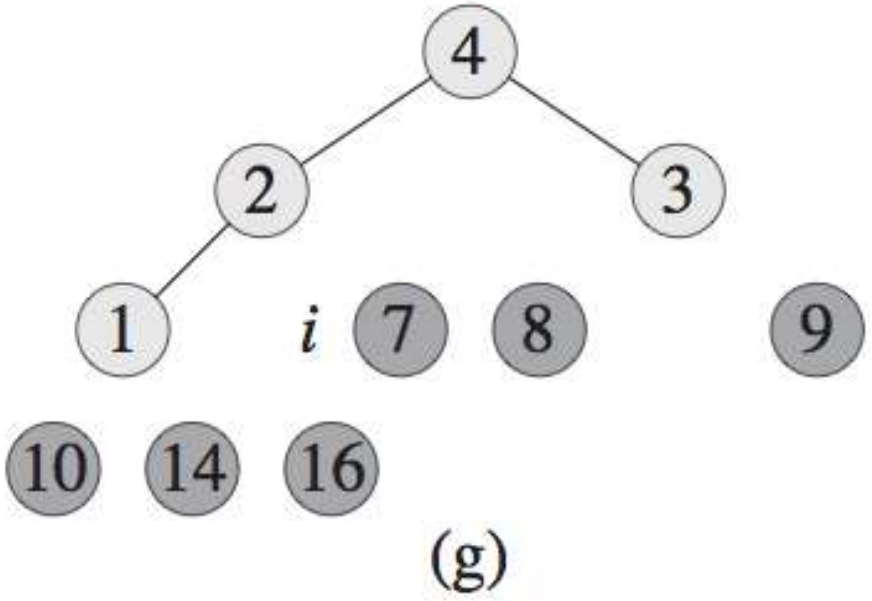
HEAPSORT(A):



(f)

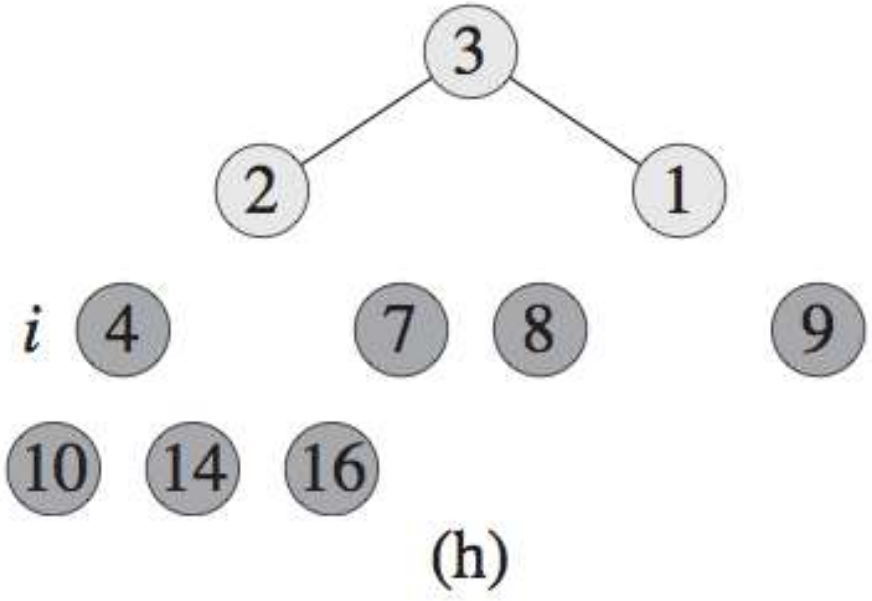
Heap Sort

HEAPSORT(A):



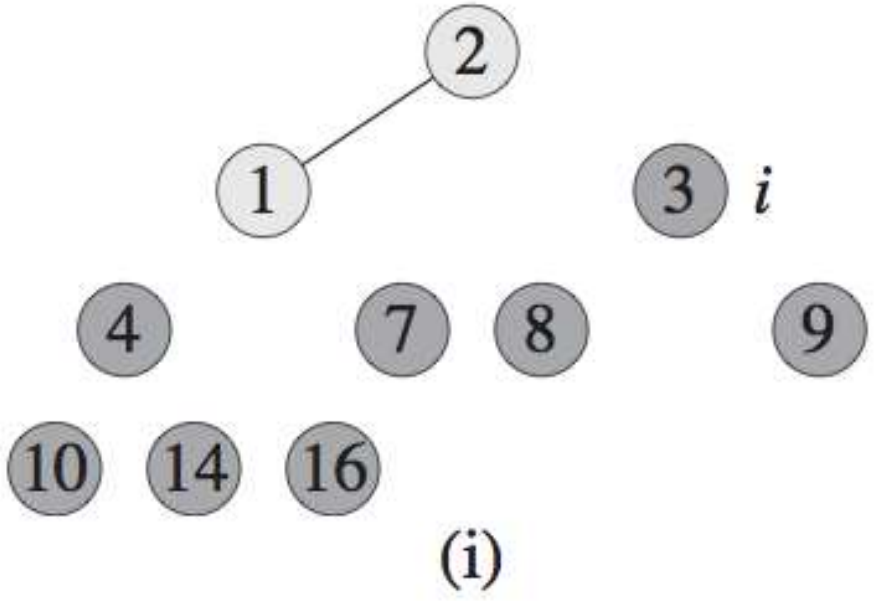
Heap Sort

HEAPSORT(A):



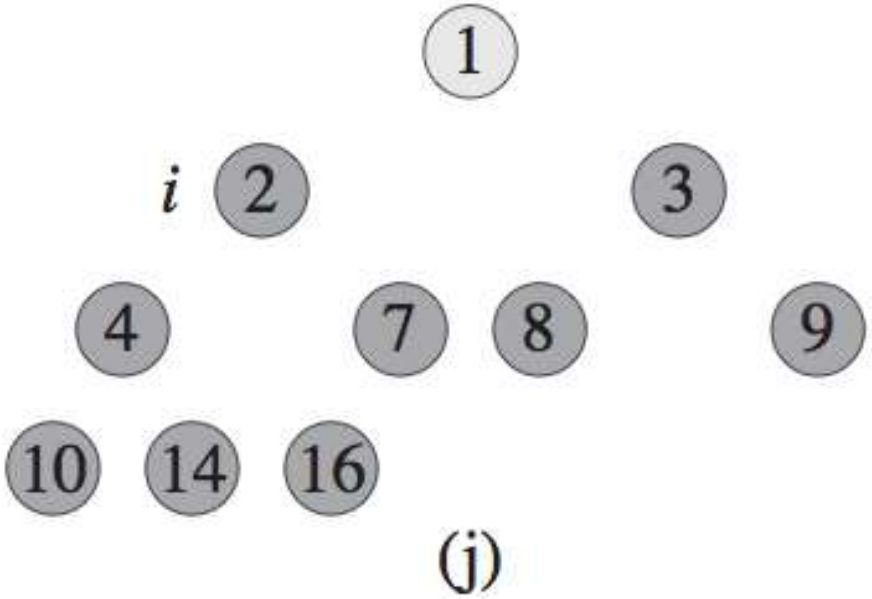
Heap Sort

HEAPSORT(A):



Heap Sort

HEAPSORT(A):



Heap Sort

HEAPSORT(A):

A	1	2	3	4	7	8	9	10	14	16