Análise e Síntese de Algoritmos

Teorema Mestre. Divide & Conquer.

CLRS Cap. 4

(continuação)

Instituto Superior Técnico

2022/2023

Exemplo 4: Multiplicação de matrizes

$$A \times B = \begin{bmatrix} a_{11} & a_{12} \\ a_{21} & a_{22} \end{bmatrix} \times \begin{bmatrix} b_{11} & b_{12} \\ b_{21} & b_{22} \end{bmatrix} = \begin{bmatrix} a_{11}b_{11} + a_{12}b_{21} & a_{11}b_{12} + a_{12}b_{22} \\ a_{21}b_{11} + a_{22}b_{21} & a_{21}b_{12} + a_{22}b_{22} \end{bmatrix}$$

Matrix-Multiplication(A, B)

```
n = rows(A)
for i ← 1 to n do
   for j ← 1 to n do
      c_ij = 0
      for k = 1 to n do
         c_ij ← c_ij + a_ik b_kj
      end for
   end for
end for
```

Algoritmo habitual

Complexidade: $\Theta(n^3)$

Exemplo 4: Multiplicação de matrizes

$$C = A \times B$$

$$\begin{bmatrix} c_{11} & c_{12} & c_{13} & c_{14} \\ c_{21} & c_{22} & c_{23} & c_{24} \\ c_{31} & c_{32} & c_{33} & c_{34} \\ c_{41} & c_{42} & c_{43} & c_{44} \end{bmatrix} = \begin{bmatrix} a_{11} & a_{12} & a_{13} & a_{14} \\ a_{21} & a_{22} & a_{23} & a_{24} \\ a_{31} & a_{32} & a_{33} & a_{34} \\ a_{41} & a_{42} & a_{43} & a_{44} \end{bmatrix} \times \begin{bmatrix} b_{11} & b_{12} & b_{13} & b_{14} \\ b_{21} & b_{22} & b_{23} & b_{24} \\ b_{31} & b_{32} & b_{33} & b_{34} \\ b_{41} & b_{42} & b_{43} & b_{44} \end{bmatrix}$$

$$\begin{matrix} \begin{bmatrix} c_{11} & c_{12} \\ c_{21} & c_{22} \end{bmatrix} & \begin{bmatrix} c_{13} & c_{14} \\ c_{23} & c_{24} \end{bmatrix} \\ \begin{bmatrix} c_{31} & c_{32} \\ c_{41} & c_{42} \end{bmatrix} & \begin{bmatrix} c_{33} & c_{34} \\ c_{43} & c_{44} \end{bmatrix} \end{matrix} = \begin{matrix} \begin{bmatrix} a_{11} & a_{12} \\ a_{21} & a_{22} \end{bmatrix} & \begin{bmatrix} a_{13} & a_{14} \\ a_{23} & a_{24} \end{bmatrix} \\ \begin{bmatrix} a_{31} & a_{32} \\ a_{41} & a_{42} \end{bmatrix} & \begin{bmatrix} a_{33} & a_{34} \\ a_{43} & a_{44} \end{bmatrix} \end{matrix} \times \begin{matrix} \begin{bmatrix} b_{11} & b_{12} \\ b_{21} & b_{22} \end{bmatrix} & \begin{bmatrix} b_{13} & b_{14} \\ b_{23} & b_{24} \end{bmatrix} \\ \begin{bmatrix} b_{31} & b_{32} \\ b_{41} & b_{42} \end{bmatrix} & \begin{bmatrix} b_{33} & b_{34} \\ b_{43} & b_{44} \end{bmatrix} \end{matrix}$$

$$\begin{bmatrix} c_{11} & c_{12} \\ c_{21} & c_{22} \end{bmatrix} = \begin{bmatrix} a_{11} & a_{12} \\ a_{21} & a_{22} \end{bmatrix} \times \begin{bmatrix} b_{11} & b_{12} \\ b_{21} & b_{22} \end{bmatrix} + \begin{bmatrix} a_{13} & a_{14} \\ a_{23} & a_{24} \end{bmatrix} \times \begin{bmatrix} b_{31} & b_{32} \\ b_{41} & b_{42} \end{bmatrix}$$

Posso multiplicar as matrizes em blocos (sub-matrizes).

Exemplo 4: Multiplicação de matrizes

$$\begin{matrix} \begin{bmatrix} a_{11} & a_{12} \\ a_{21} & a_{22} \end{bmatrix} & \begin{bmatrix} a_{13} & a_{14} \\ a_{23} & a_{24} \end{bmatrix} \\ \begin{bmatrix} a_{31} & a_{32} \\ a_{41} & a_{42} \end{bmatrix} & \begin{bmatrix} a_{33} & a_{34} \\ a_{43} & a_{44} \end{bmatrix} \end{matrix} \times \begin{matrix} \begin{bmatrix} b_{11} & b_{12} \\ b_{21} & b_{22} \end{bmatrix} & \begin{bmatrix} b_{13} & b_{14} \\ b_{23} & b_{24} \end{bmatrix} \\ \begin{bmatrix} b_{31} & b_{32} \\ b_{41} & b_{42} \end{bmatrix} & \begin{bmatrix} b_{33} & b_{34} \\ b_{43} & b_{44} \end{bmatrix} \end{matrix}$$

$$\begin{bmatrix} a_{11} & a_{12} \\ a_{21} & a_{22} \end{bmatrix} \times \begin{bmatrix} b_{11} & b_{12} \\ b_{21} & b_{22} \end{bmatrix} + \begin{bmatrix} a_{13} & a_{14} \\ a_{23} & a_{24} \end{bmatrix} \times \begin{bmatrix} b_{31} & b_{32} \\ b_{41} & b_{42} \end{bmatrix}$$

$$\begin{bmatrix} a_{11} & a_{12} \\ a_{21} & a_{22} \end{bmatrix} \times \begin{bmatrix} b_{13} & b_{14} \\ b_{23} & b_{24} \end{bmatrix} + \begin{bmatrix} a_{13} & a_{14} \\ a_{23} & a_{24} \end{bmatrix} \times \begin{bmatrix} b_{33} & b_{34} \\ b_{43} & b_{44} \end{bmatrix}$$

$$\begin{bmatrix} a_{31} & a_{32} \\ a_{41} & a_{42} \end{bmatrix} \times \begin{bmatrix} b_{11} & b_{12} \\ b_{21} & b_{22} \end{bmatrix} + \begin{bmatrix} a_{33} & a_{34} \\ a_{43} & a_{44} \end{bmatrix} \times \begin{bmatrix} b_{31} & b_{32} \\ b_{41} & b_{42} \end{bmatrix}$$

$$\begin{bmatrix} a_{31} & a_{32} \\ a_{41} & a_{42} \end{bmatrix} \times \begin{bmatrix} b_{13} & b_{14} \\ b_{23} & b_{24} \end{bmatrix} + \begin{bmatrix} a_{33} & a_{34} \\ a_{43} & a_{44} \end{bmatrix} \times \begin{bmatrix} b_{33} & b_{34} \\ b_{43} & b_{44} \end{bmatrix}$$

$$\begin{matrix} a_{31} \times b_{13} + a_{32} \times b_{23} \\ a_{31} \times b_{14} + a_{32} \times b_{24} \\ a_{41} \times b_{13} + a_{42} \times b_{23} \\ a_{41} \times b_{14} + a_{42} \times b_{24} \end{matrix}$$

Algoritmo recursivo

Multiplicação de matrizes recursiva:

- Partir cada matriz $n \times n$ em 4 matrizes, cada uma com dimensão $n/2 \times n/2$, até as matrizes terem dimensão $n = 1$.
- Efetuar *bottom-up* a multiplicação de 2 matrizes $n \times n$ através de 8 multiplicações de matrizes $n/2 \times n/2$ (e mais 4 somas).

$T(n) = 8T(n/2) + \Theta(n^2)$

- $a = 8, b = 2, d = 2$
- $d = 2$ is $<$ than $\log_2 8$
- $T(n) = \Theta(n^{\log_b a}) = \Theta(n^3)$ (caso 1 do Teorema Mestre)

Não é melhor que o algoritmo habitual!

$$\begin{bmatrix} A & B \\ C & D \end{bmatrix} \times \begin{bmatrix} E & F \\ G & H \end{bmatrix} = \begin{bmatrix} A \times E + B \times G & A \times F + B \times H \\ C \times E + D \times G & C \times F + D \times H \end{bmatrix}$$

Algoritmo de Strassen

$M_1 = A \times (F - H)$
$M_2 = H \times (A + B)$
$M_3 = E \times (C + D)$
$M_4 = D \times (G - E)$
$M_5 = (A + D) \times (E + H)$
$M_6 = (B - D) \times (G + H)$
$M_7 = (A - C) \times (E + F)$

$A \times E + B \times G = M_5 + M_4 - M_2 + M_6$
$A \times F + B \times H = M_1 + M_2$
$C \times E + D \times G = M_3 + M_4$
$C \times F + D \times H = M_5 + M_1 - M_3 - M_7$

Permite reduzir o número de multiplicações de 8 para 7, à custa de introduzir um número (constante) de adições.

$T(n) = 7T(n/2) + \Theta(n^2)$

- $a = 7, b = 2, d = 2$
- $d = 2$ is $<$ than $\log_2 7$
- $T(n) = \Theta(n^{\log_b a}) = \Theta(n^{2.81})$ (caso 1 do Teorema Mestre)

Algoritmo habitual (3 ciclos): $\Theta(n^3)$

Algoritmo recursivo: $T(n) = 8T(n/2) + \Theta(n^2) = \Theta(n^3)$

Algoritmo de Strassen: $T(n) = 7T(n/2) + \Theta(n^2) = \Theta(n^{2.81})$

Análise e Síntese de Algoritmos

Amontoados. Heapsort.

CLRS Cap. 6

Instituto Superior Técnico

2022/2023

Contexto

- Revisão [CLRS, Cap.1-13]
  - Fundamentos; notação; exemplos
- Técnicas de Síntese de Algoritmos [CLRS, Cap.15-16]
  - Programação dinâmica
  - Algoritmos greedy
- Algoritmos em Grafos [CLRS, Cap.21-26]
  - Algoritmos elementares
  - Caminhos mais curtos
  - Fluxos máximos
  - Árvores abrangentes
- Programação Linear [CLRS, Cap.29]
  - Algoritmos e modelação de problemas com restrições lineares
- Tópicos Adicionais [CLRS, Cap.32-35]
  - Complexidade Computacional

Amontoados: Propriedades

Vetor de valores interpretado como uma árvore binária (essencialmente completa)

- *length*$[A]$: tamanho do vetor
- *heap-size*$[A]$: número de elementos no amontoado
- $A[1]$: raiz da árvore

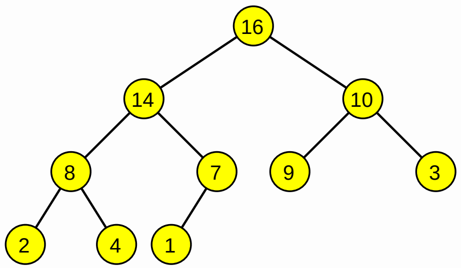
Relações entre nós da árvore

- $\text{Parent}(i) = \lfloor i/2 \rfloor$
- $\text{Left}(i) = 2i$
- $\text{Right}(i) = 2i + 1$

Propriedade de amontoado

- $A[\text{Parent}(i)] \geq A[i]$

Amontoados: Exemplo



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

Amontoados

Árvore binária completa

- Cada nó tem exatamente 0 ou 2 filhos
- Cada folha (i.e. nó com 0 filhos) tem a mesma profundidade d

Árvore binária essencialmente completa

- Cada nó pode ter 0, 1 ou 2 filhos
  - Se um nó tiver apenas um filho, será o filho do lado esquerdo
- Cada folha pode ter profundidade d ou $d - 1$
  - Qualquer nó interno à profundidade $d - 1$, posicionado à esquerda de qualquer nó folha à mesma profundidade

Profundidade: número de nós entre a raiz e um dado nó

Amontoados

Amontoado "Heap"
Vetor A de valores interpretado como uma árvore binária (essencialmente) completa

Propriedades

- $A[1]$: raiz da árvore ($i = 1$) — (A[0] se $i = 0$)
- $length(A)$: tamanho do vetor
- $heap\text{-}size(A)$: número de elementos do amontoado
- $\text{Parent}(i) = \lfloor i/2 \rfloor$ — ($\lceil i/2 \rceil - 1$ se $i = 0$)
- $\text{Left}(i) = 2\,i$ — ($2\,i + 1$ se $i = 0$)
- $\text{Right}(i) = 2\,i + 1$ — ($2\,i + 2$ se $i = 0$)

Amontoados

Invariante min-heap
O valor do nó i é sempre menor ou igual ao valor dos nós descendentes

- $A[\text{Parent}(i)] \leq A[i]$
- $A[i] \leq A[\text{Left}(i)] \ \wedge \ A[i] \leq A[\text{Right}(i)]$

Aplicação: Usado na implementação de priority queues

Invariante max-heap
O valor do nó i é sempre maior ou igual ao valor dos nós descendentes

- $A[\text{Parent(i)}] \geq A[i]$
- $A[i] \geq A[\text{Left}(i)] \ \wedge \ A[i] \geq A[\text{Right}(i)]$

Aplicação: Usado na implementação do Heapsort

Amontoados

Exercício: max-heap ou min-heap?

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

$\rightarrow$ max-heap

	1	2	3	4	5	6
A	1	3	8	5	9	6

$\rightarrow$ nenhum

	1	2	3	4	5	6	7	8
A	1	2	3	6	9	5	10	14

$\rightarrow$ min-heap

Heapsort

Operações mantendo invariante max-heap*

- Heap-Maximum(A): devolve o valor máximo de A
- Max-Heapify(A, i): corrige uma violação da invariante em i
  - Assume a invariante em Left(i) e Right(i)
- Build-Max-Heap(A): constroi um max-heap a partir de um vetor arbitrário

* Análogo para min-heap

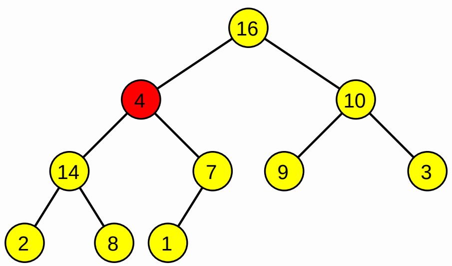
Heap-Maximum(A)
return $A[1]$

Complexidade

- $O(1)$

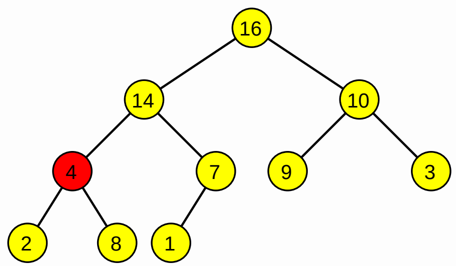
Max-Heapify(A, i)
$l \leftarrow$ Left(i)
$r \leftarrow$ Right(i)
$largest \leftarrow i$
if $l \leq heap\text{-}size(A) \;\wedge\; A[l] > A[i]$ **then**
largest $\leftarrow$ l
end if
if r $\leq$ heap-size(A) && A[r] ¿ A[largest] **then**
$largest \leftarrow r$
end if
if $largest \neq i$ **then**
exchange $A[i] \leftrightarrow A[largest]$
Max-Heapify$(A, largest)$
end if

Exemplo
Max-Heapify$(A, 2)$



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

Exemplo
Max-Heapify$(A, 2) \rightarrow$ Max-Heapify$(A, 4)$

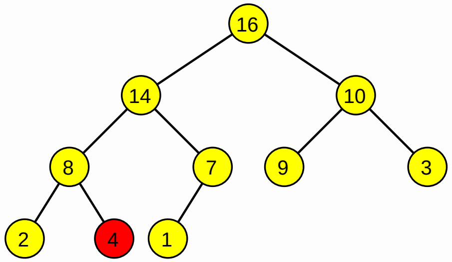


1	2	3	4	5	6	7	8	9	10
16	14	10	4	7	9	3	2	8	1

Heapsort

Exemplo

Max-Heapify$(A, 2)$ → Max-Heapify$(A, 4)$ → Max-Heapify$(A, 9)$



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

Heapsort

Complexidade

- Altura do amontoado: $h = \lfloor \log\ n \rfloor$ (Árvore binária)
- Complexidade de Max-Heapify: $O(\log\ n)$

Recorrência: $T(n) = T(2n/3) + \Theta(1)$

- $a = 1, b = 3/2, d = 0$
- $d = 0$ is $=$ than $\log_{3/2} 1$
- $T(n) = \Theta(n^d \log\ n) = \Theta(\log\ n)$ (caso 2 do Teorema Mestre)

Heapsort

Build-Max-Heap(A)

for $i \leftarrow \lfloor \textit{heap-size}(A)/2 \rfloor$ downto 1 **do**
  Max-Heapify(A, i)
end for

Questão:

- Porque é que se inicia a $\lfloor \textit{heap-size}(A)/2 \rfloor$?
  Os elementos com índices $\lfloor \textit{heap-size}(A)/2 \rfloor + 1\ \ldots\ \textit{heap-size}(A)$ são folhas e portanto já são max-heaps com 1 elemento

Heapsort

Build-Max-Heap(A)

for $i \leftarrow \lfloor \textit{heap-size}(A)/2 \rfloor$ downto 1 **do**
  Max-Heapify(A, i)
end for

Complexidade

- Análise simples: $O(n\ \log\ n)$
- Possível provar: $O(n)$
  - O tempo de execução de Max-Heapify depende da altura (distância à folha mais longe) do nó onde está a ser aplicado
  - A altura da maioria dos nós é pequena, muito inferior a $\log n$

Heapsort

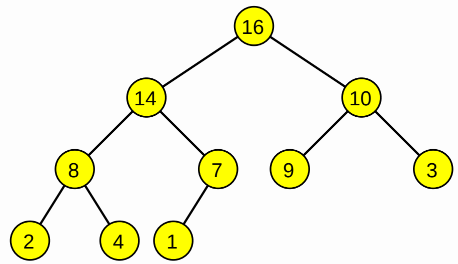
Heapsort(A)
Build-Max-Heap(A)
for $i \leftarrow \lfloor length(A) \rfloor$ downto 2 **do**
exchange $A[1] \leftrightarrow A[i]$
$heap\text{-}size(A) \leftarrow heap\text{-}size(A) - 1$
Max-Heapify($A, 1$)
end for

Intuição

- Extrair consecutivamente o elemento máximo de um amontoado
- Colocar esse elemento na posição (certa) do vetor

Heapsort

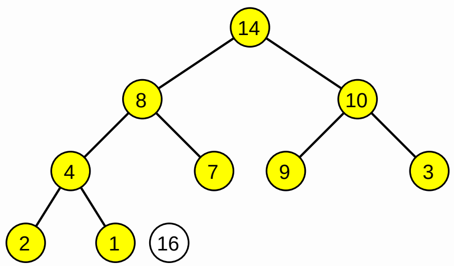
Exemplo: Heapsort(A)



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

Heapsort

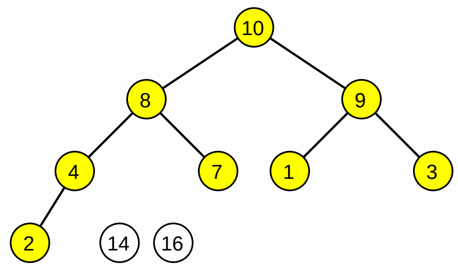
Exemplo: Heapsort(A)



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

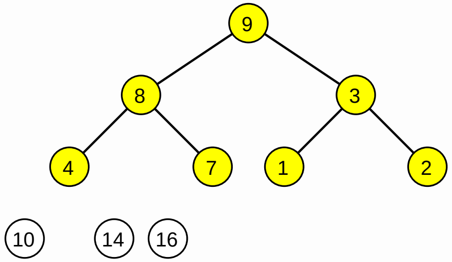
Heapsort

Exemplo: Heapsort(A)



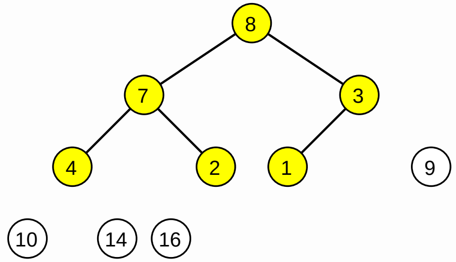
1	2	3	4	5	6	7	8	9	10
10	8	9	4	7	1	3	2	14	16

Exemplo: Heapsort(A)



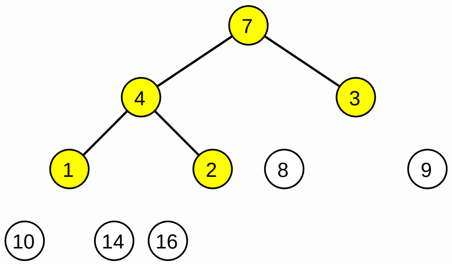
1	2	3	4	5	6	7	8	9	10
9	8	3	4	7	1	2	10	14	16

Exemplo: Heapsort(A)



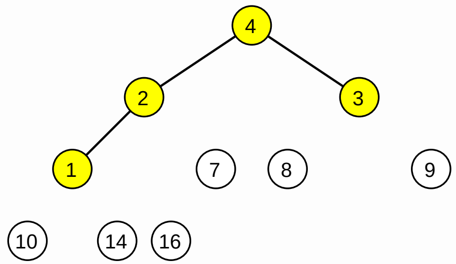
1	2	3	4	5	6	7	8	9	10
8	7	3	4	2	1	9	10	14	16

Exemplo: Heapsort(A)



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

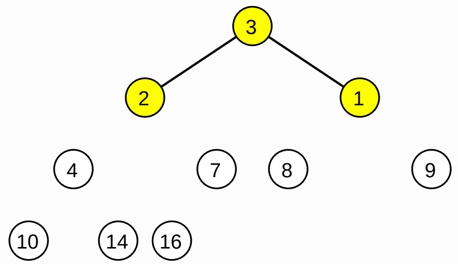
Exemplo: Heapsort(A)



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

Heapsort

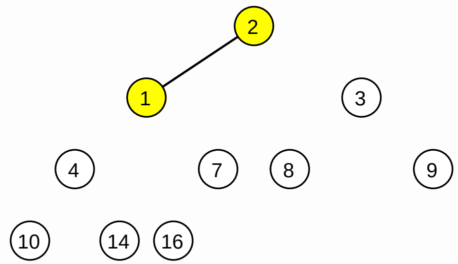
Exemplo: Heapsort(A)



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

Heapsort

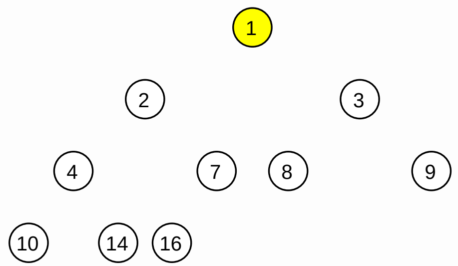
Exemplo: Heapsort(A)



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

Heapsort

Exemplo: Heapsort(A)



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

Heapsort

Heapsort(A)
  Build-Max-Heap(A)
  for $i \leftarrow \lfloor length(A) \rfloor$ downto 2 **do**
    $A[1] \leftrightarrow A[i]$
    $heap\text{-}size(A) \leftarrow heap\text{-}size(A) - 1$
    Max-Heapify($A, 1$)
  end for

Complexidade

- $O(n \log n)$

Fila de prioridades (FIFO)
Implementa um conjunto de elementos S, em que cada um dos elementos tem associada um valor/prioridade

Exemplos

- Filas nas finanças
- Escalonamento de processos num computador partilhado
- Reencaminhamento de pacotes na rede
- ...

Para manipularmos a fila de prioridades, necessitamos de um conjunto de operações.

Operações

- Max-Heap-Insert(S, x) - insere o elemento x no conjunto S
- Heap-Maximum(S) - devolve o elemento de S com o valor máximo
- Heap-Extract-Max(S) - remove e devolve o elemento de S com o valor máximo
- Heap-Increase-Key(S, x, k) - incrementa o valor de x com o valor k

De forma a implementarmos estas operações de forma eficiente !
$\Rightarrow$ utilizamos um Amontoado (Heap)

Heap-Maximum(A)
  return $A[1]$

Complexidade

- $O(1)$

Heap-Extract-Max(A)
  $max \leftarrow A[1]$
  $A[1] \leftarrow A[heap\text{-}size[A]]$
  $heap\text{-}size[A] \leftarrow heap\text{-}size[A] - 1$
  Max-Heapify($A, 1$)
  return max

Complexidade

- $O(\log\ n)$

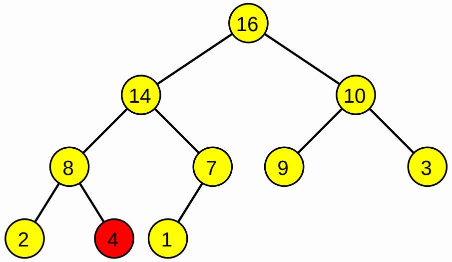
Heap-Increase-Key(A, i, key)
  if $key < A[i]$ **then**
    error "new key is smaller than current key"
  end if
  $A[i] \leftarrow key$
  while $i > 1 \wedge A[\text{Parent}(i)] < A[i]$ **do**
    $A[i] \leftrightarrow A[\text{Parent}(i)]$
    $i \leftarrow \text{Parent}(i)$
  end while

Complexidade

- $O(\log\ n)$

Priority queues

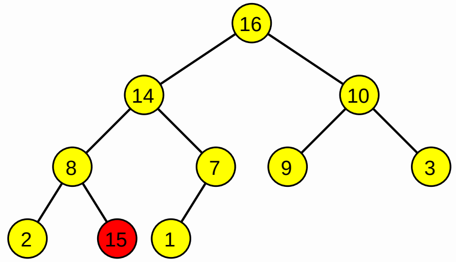
Exemplo: Heap-Increase-Key(A, i, key)



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

Priority queues

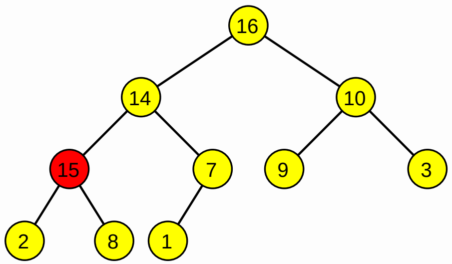
Exemplo: Heap-Increase-Key(A, i, key)



1	2	3	4	5	6	7	8	9	10
16	14	10	8	7	9	3	2	15	1

Priority queues

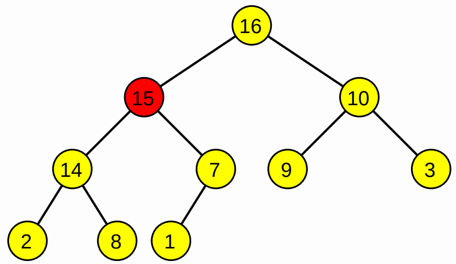
Exemplo: Heap-Increase-Key(A, i, key)



1	2	3	4	5	6	7	8	9	10
16	14	10	15	7	9	3	2	8	1

Priority queues

Exemplo: Heap-Increase-Key(A, i, key)



1	2	3	4	5	6	7	8	9	10
16	15	10	14	7	9	3	2	8	1

Priority queues

Max-Heap-Insert(A, key)

$heap\text{-}size[A] \leftarrow heap - size[A] + 1$
$A[heap\text{-}size[A]] \leftarrow -\infty$
Heap-Increase-Key$(A, heap\text{-}size[A], key)$

Complexidade

- $O(\log n)$