Análise e Síntese de Algoritmos

Amontoados. Heapsort. Priority Queues.

CLRS Cap. 6

Prof. Pedro T. Monteiro

IST - Universidade de Lisboa

2024/2025

Contexto

- Revisão [CLRS, Cap.1-13]
  - Fundamentos; notação; exemplos
- Técnicas de Síntese de Algoritmos [CLRS, Cap.15-16]
  - Programação dinâmica [CLRS, Cap.15]
  - Algoritmos greedy [CLRS, Cap.16]
- Algoritmos em Grafos [CLRS, Cap.21-26]
  - Algoritmos elementares
  - Caminhos mais curtos [CLRS, Cap.22,24-25]
  - Árvores abrangentes [CLRS, Cap.23]
  - Fluxos máximos [CLRS, Cap.26]
- Programação Linear [CLRS, Cap.29]
  - Algoritmos e modelação de problemas com restrições lineares
- Tópicos Adicionais
  - Emparelhamento de Cadeias de Caracteres [CLRS, Cap.32]
  - Complexidade Computacional [CLRS, Cap.34]

Resumo

Amontoados

Árvore binária completa

- Cada nó interno (+ raíz) tem 2 descendentes
- Cada folha tem:
  - 0 descendentes
  - mesma profundidade d

Árvore binária essencialmente completa

- Cada nó interno (+ raíz) tem 2 descendentes:
  - Excepção: nó a profundidade $d-1$ sem descendente direito
- Cada folha tem:
  - 0 descendentes
  - profundidade d ou $d-1$

Amontoados

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

Propriedades

- `A[1]` - raíz da árvore ($i = 1$) (`A[0]` se $i = 0$)
- `length(A)` - tamanho do vector
- `heap-size(A)` - número de elementos do amontado
- `parent(i)` - $\lfloor i/2 \rfloor$ ($\lceil i/2 \rceil - 1$ se $i = 0$)
- `left(i)` - $2\,i$ ($2\,i + 1$ se $i = 0$)
- `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[parent(i)]` $\leq$ `A[i]`
`A[i]` $\leq$ `A[left(i)]` `&&` `A[i]` $\leq$ `A[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[parent(i)]` $\geq$ `A[i]`
`A[i]` $\geq$ `A[left(i)]` `&&` `A[i]` $\geq$ `A[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*

- `max(A)` - retorna o valor máximo de A
- `max-heapify(A, i)` - corrige uma violação da invariante em i
  Pressuposto: assume a invariante em `left(i)` e `right(i)`
- `build-max-heap(A)` - constroi um Max-heap a partir de um vector desordenado

* Análogo para Min-heap

Heapsort

```
max(A)
  return A[1]
```

Complexidade

- $O(1)$

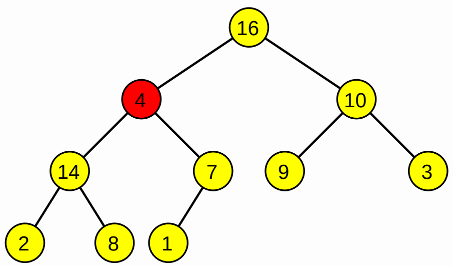
Heapsort

```
max-heapify(A, i)
  l ← left(i)
  r ← right(i)
  largest ← i
  if l ≤ heap-size(A) && A[l] > A[i] then
    largest ← l
  end if
  if r ≤ heap-size(A) && A[r] > A[largest] then
    largest ← r
  end if
  if largest ≠ i then
    exchange A[i] ↔ A[largest]
    max-heapify(A, largest)
  end if
```


Heapsort

Exemplo

`max-heapify(A, 2)`

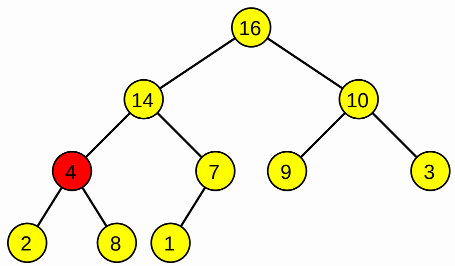


1	2	3	4	5	6	7	8	9	10
16	4	10	14	7	9	3	2	8	1

Heapsort

Exemplo

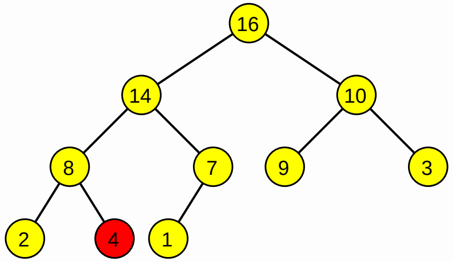
`max-heapify(A, 2)` → `max-heapify(A, 4)`



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

Exemplo

`max-heapify(A, 2)` $\to$ `max-heapify(A, 4)` $\to$ `max-heapify(A, 9)`



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

Complexidade (análise simples)

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

Complexidade (análise divide-and-conquer)

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

Maior desequilíbrio possível (pior caso): # nós numa sub-árvore limitada a $\leq \frac{2n}{3}$

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


```
build-max-heap(A)
  for i ← ⌊ heap-size(A) / 2 ⌋ downto 1 do
    max-heapify(A, i)
  end for
```

Questões:

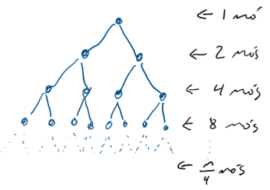
- Porque é que se inicia a `heap-size(A) / 2` ?
  `⌊ heap-size(A) / 2 ⌋ + 1 ... heap-size(A)` são max-heaps
- Porque é que se vai `downto 1` ?
  cada `max-heapify(A, i)` satisfaz o pressuposto


```
build-max-heap(A)
  for i ← ⌊ heap-size(A) / 2 ⌋ downto 1 do
    max-heapify(A, i)
  end for
```

Complexidade

- Análise simples: $O(n\ log\ n)$
- Possível provar: $\Theta(n)$ (como?)
  - O tempo de execução de `max-heapify` depende da altura do nó a que está a ser aplicado
  - A altura maioria dos nós é pequena, muito inferior a `log n`

build-max-heap(A) - Prova complexidade $\Theta(n)$



nível	# nós
log n	1
⋮	⋮
3	$n/16$
2	$n/8$
1	$\frac{n}{4}$

Quantas operações faz o max-heapify?

- No nível 1: $\frac{n}{4}(1 \times c)$
- No nível 2: $\frac{n}{8}(2 \times c)$

⋮

c é uma constante
$\frac{n}{4}$ porque há $\frac{n}{4}$ nós
1 porque só há 1 nível

Quantas operações faz o build max heap?

$$\frac{n}{4}(1.c) + \frac{n}{8}(2.c) + \frac{n}{16}(3.c) + \dots + 1((\log_2 n).c)$$

Substituição: $\frac{n}{4} = 2^k$ → $n = 4.2^k = 2^2.2^k = 2^{k+2}$

$$2^k(1.c) + \frac{2^k}{2}(2.c) + \frac{2^k}{4}(3.c) + \dots + 1((\log_2 2^{k+2}).c)$$

$$c.2^k\underbrace{\left(\frac{1}{2^0} + \frac{2}{2^1} + \frac{3}{2^2} + \dots + \frac{k+2}{2^k}\right)}_{\text{Série convergente} \Rightarrow \text{limitado por uma constante!}}$$

c: é uma constante

2^k: $\frac{n}{4} \Rightarrow \Theta(n)$

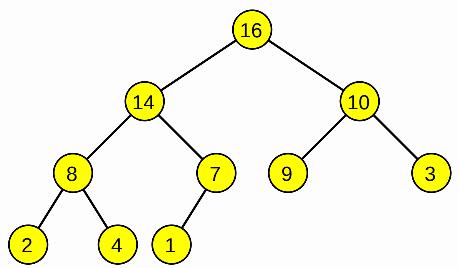
Logo, $\Theta(n)$ //


```
heapsort(A)
  build-max-heap(A)
  for i ← ⌊ length(A) ⌋ downto 2 do
    A[1] ↔ A[i]
    heap-size(A) ← heap-size(A) - 1
    max-heapify(A, 1)
  end for
```

Intuição

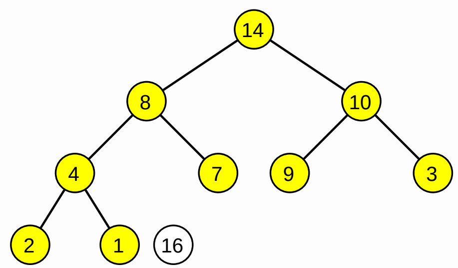
- Extrair consecutivamente o elemento máximo de um amontoado
- Colocar esse elemento na posição (certa) do vector

Exemplo: heapsort(A)



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

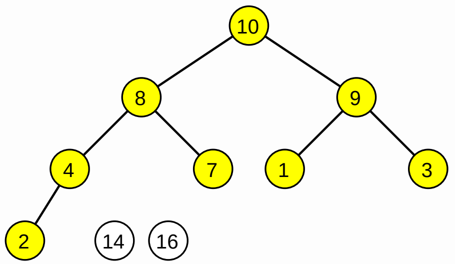
Exemplo: heapsort(A)



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

Heapsort

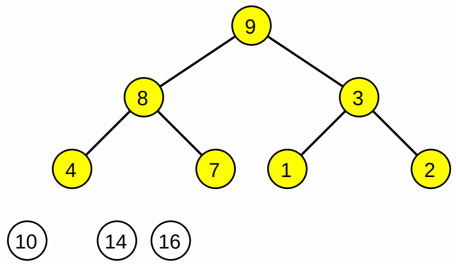
Exemplo: `heapsort(A)`



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

Heapsort

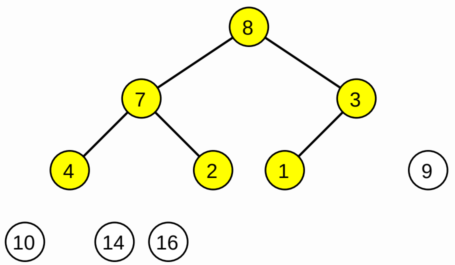
Exemplo: `heapsort(A)`



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

Heapsort

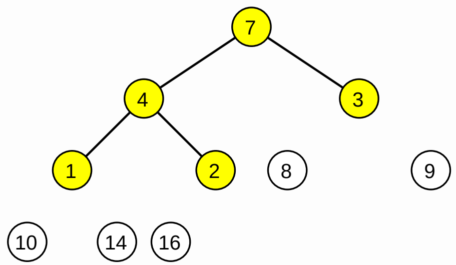
Exemplo: `heapsort(A)`



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

Heapsort

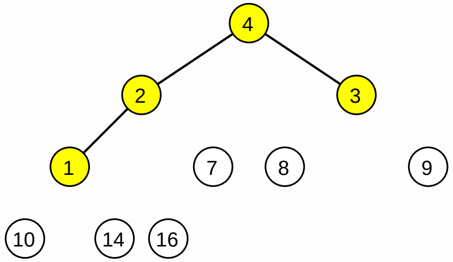
Exemplo: `heapsort(A)`



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

Heapsort

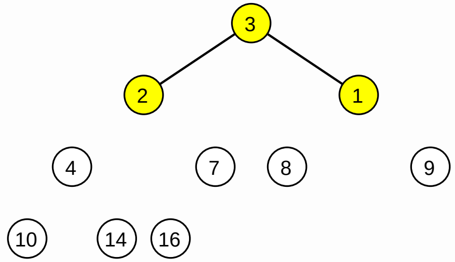
Exemplo: `heapsort(A)`



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

Heapsort

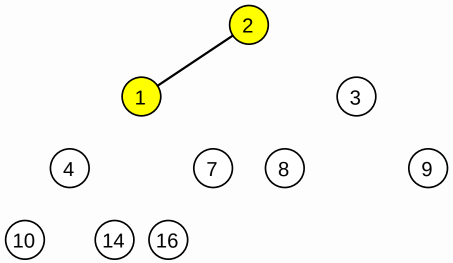
Exemplo: `heapsort(A)`



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

Heapsort

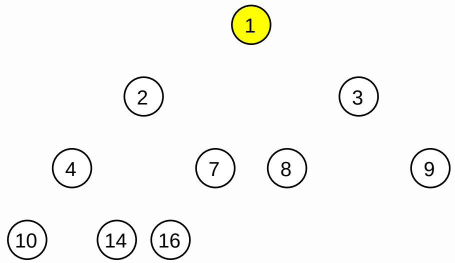
Exemplo: `heapsort(A)`



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

Heapsort

Exemplo: `heapsort(A)`



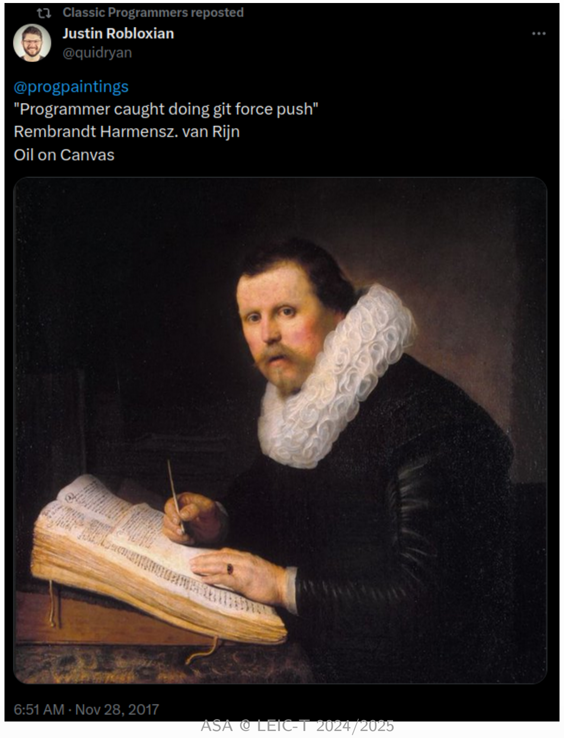
1	2	3	4	5	6	7	8	9	10
1	2	3	4	7	8	9	10	14	16

Heapsort

```
heapsort(A)
  build-max-heap(A)
  for i ← ⌊ length(A) ⌋ downto 2 do
    A[1] ↔ A[i]
    heap-size(A) ← heap-size(A) - 1
    max-heapify(A, 1)
  end for
```

Complexidade

- $O(n \log n)$



Priority queues

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
- ...

Priority queues

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)

Priority queues

```
heap-maximum(A)
  return A[1]
```

Complexidade

- $O(1)$

```
heap-extract-max(A)

  max ← A[1]
  A[1] ← A[heap-size[A]]
  heap-size[A] ← heap-size[A] -
  1
  max-heapify(A, 1)
  return max
```

Complexidade

- $O(log\ n)$

Priority queues

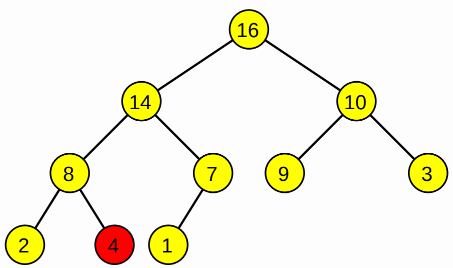
```
heap-increase-key(A, i, key)
  if key < A[i] then
    error "new key is smaller than current key"
  end if
  A[i] ← key
  while i > 1 and A[parent(i)] < A[i] do
    A[i] ↔ A[parent(i)]
    i ← parent(i)
  end while
```

Complexidade

- $O(log\ n)$

Priority queues

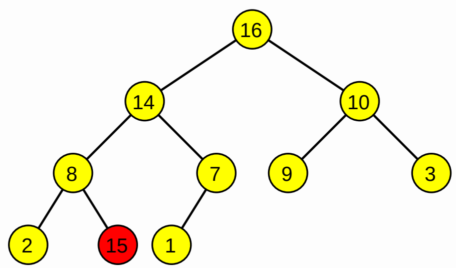
Exemplo: `heap-increase-key(A, i, key)`



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

Priority queues

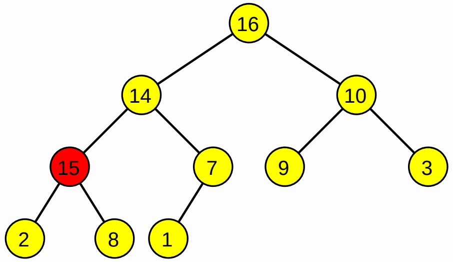
Exemplo: `heap-increase-key(A, i, key)`



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

Priority queues

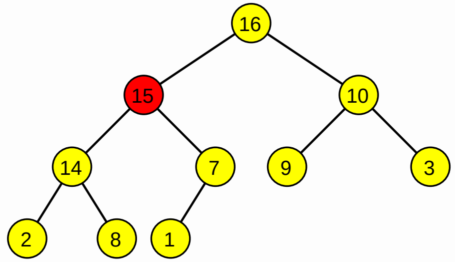
Exemplo: `heap-increase-key(A, i, key)`



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-size[A] ← heap-size[A] + 1
  A[heap-size[A]] ← −∞
  heap-increase-key(A, heap-size[A], key)
```

Complexidade

- $O(log\ n)$

Questão:

- Porque é que se inicializa a $-\infty$?
  Guarda do `heap-increase-key`

Priority queues

Algoritmos que usam filas de prioridade

- Algoritmo de Dijkstra - Caminhos mais curtos
- Algoritmo de Prim - Árvores abrangentes
- Algoritmo de Huffman - Codificação de dados
- ...

Dúvidas?