Algoritmos e estruturas de Dados

Heaps

Prof. Maiquel de Brito

Filas de prioridades - problemática

Uma tarefa comum em programas de computador é selecionar um dado entre vários de acordo com alguma prioridade (ex. menor valor, maior valor etc)

Há várias formas de implementar filas de prioridades. Ex:

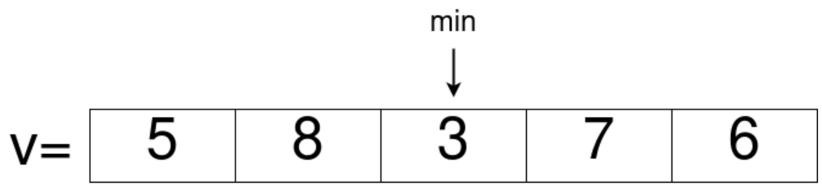
- armazenar dados em uma lista e manter um ponteiro para o menor elemento;
- armazenar os dados de forma ordenada.

Em todas essas estratégias, há operações que tem complexidade *O(n)* (cf.próximos slides)

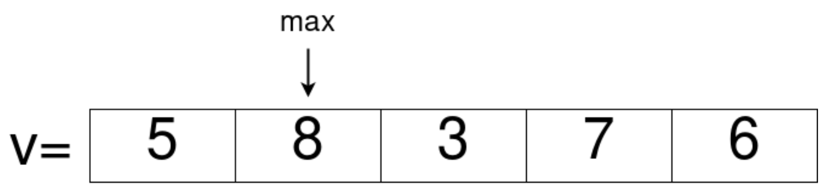
Filas de prioridades - problemática

Estratégia 1:

armazenar dados em uma lista e manter um ponteiro para o menor (ou maior) elemento



ou



Vantagem: o menor (ou maior) elemento é acessado em *O(1)*

Desvantagem: ao remover um elemento, deve-se atualizar o indicador *min* (ou *max*). Para isso, deve-se avaliar todos os elementos em *O(n).*

Filas de prioridades - problemática

Estratégia 2:

- armazenar os dados de forma ordenada

v=

3	5	6	7	8

ou

v=

8	7	6	5	3

Vantagem: o menor elemento é o primeiro e pode ser acessado em *O(1)*

Desvantagem: ao inserir um novo elemento, deve-se procurar a sua posição correta. Isso pode ser feito em *O(log n)* usando pesquisa binária. Mas, ao fazer a inserção, todos os demais elementos devem ser deslocados à direita, com custo *O*(*n*) no pior caso.

Heaps

Heaps são estruturas que permitem a implementação de filas de prioridade com operações de complexidade máxima *O(log n)*

Há dois tipos de heap:

- *min_heap:* o elemento prioritário é aquele que tem menor valor
- *max_heap*: o elemento prioritário é aquele que tem maior valor

Heaps

Formalmente:

Um *min_heap* é uma sequência *v* com *n* elementos indexado por *i=1, …, n*, em que, para *i* ≤ 2 ≤ n:

$$v[\lfloor i/2 \rfloor] \leq v[i]$$

Exemplo:

i	1	2	3	4	5	6	7
v=	1	2	5	10	3	7	11

v[1] ≤ v[2] v[2] ≤ v[4] v[3] ≤ v[6]

v[1] ≤ v[3] v[2] ≤ v[5] v[3] ≤ v[7]

Heaps

Formalmente:

Um *max_heap* é uma sequência *v* com *n* elementos indexado por $i=1, \ldots, n$, em que, para $i \leq 2 \leq n$:

$$v[\lfloor i/2 \rfloor] \geq v[i]$$

Exemplo:

i	1	2	3	4	5	6	7
v=	11	7	10	5	2	1	3

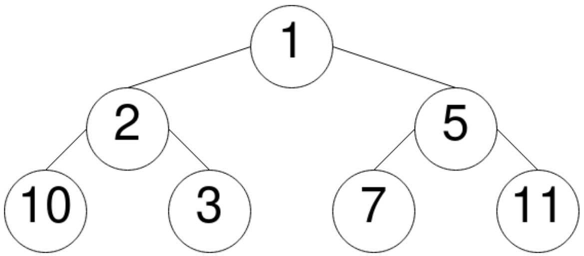
$v[1] \geq v[2]$ $v[2] \geq v[4]$ $v[3] \geq v[6]$

$v[1] \geq v[3]$ $v[2] \geq v[5]$ $v[3] \geq v[7]$

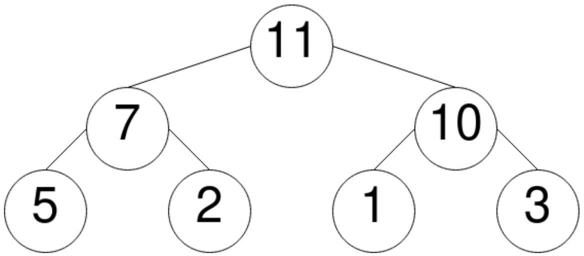
Heaps

Heaps podem ser visualizados na forma de árvores binárias de forma que, para um elemento para o elemento *v[i]* terá *v[2i]* como seu filho à esquerda e *v[2i+1]* como seu filho à direita.

i	1	2	3	4	5	6	7
v=	1	2	5	10	3	7	11

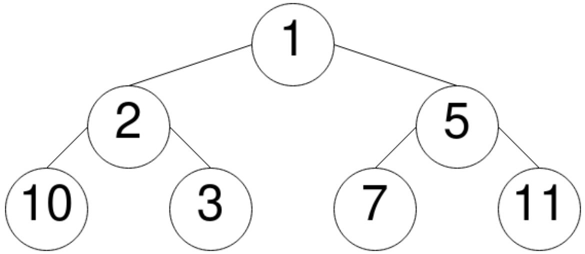


i	1	2	3	4	5	6	7
v=	11	7	10	5	2	1	3



Heaps

i	1	2	3	4	5	6	7
v=	11	7	10	5	2	7	3



Heaps podem ser armazenados na forma de árvores.

Também podem ser armazenados na forma de alguma estrutura sequencial, como vetores ou listas.

O armazenamento em vetores é uma abordagem comum pois otimiza o uso de memória ao evitar o uso de ponteiros *(Kleinberg e Tardos, 2006)*. A limitação do tamanho fixo dos vetores pode ser tratada com alocação dinâmica de memória.

Mesmo ao utilizar vetores, considera-se que os elementos de um heap de tamanho *n* são indexados de *1* a *n*, enquanto os elementos do vetor que contém o heap são indexados de *0* a *n-1.*

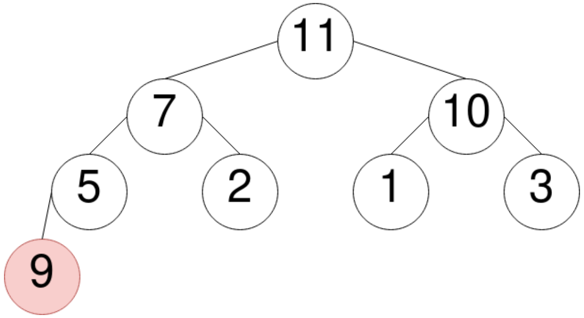
Operações em Heaps

A primeira posição de um *heap* será ocupada pelo menor elemento no caso de *min_heap* e pelo maior elemento no caso de *max_heap.* Assim, esses elementos podem ser recuperados em *O(1).*

Novos elementos podem ser adicionados ao final do vetor . No entanto, isso pode violar a propriedade de *heap* da estrutura.

Por exemplo, no *max_heap* abaixo, ao adicionar-se o elemento 9 na posição 8, tem-se $v[4]<v[8]$

i	1	2	3	4	5	6	7	8
v=	11	7	10	5	2	1	3	9

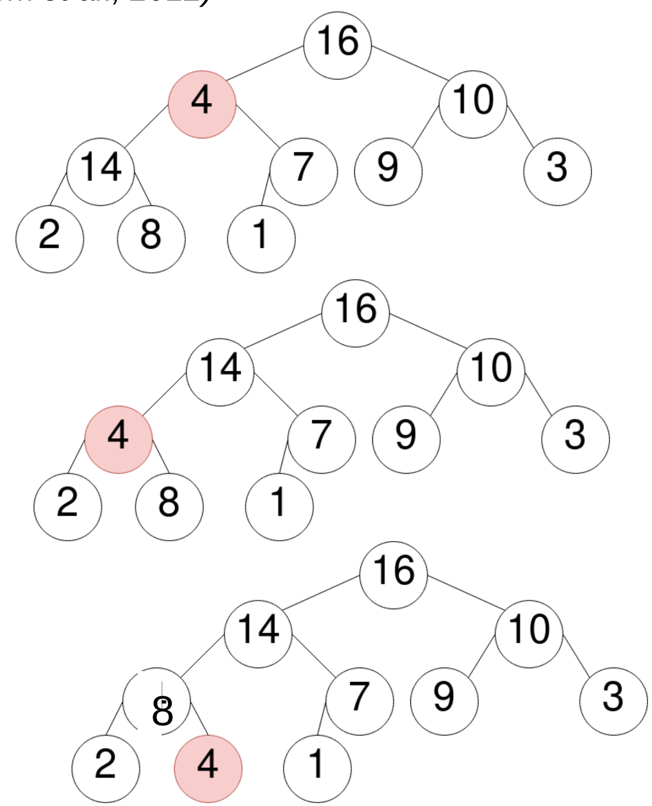


Para resolver este problema, aplica-se uma operação sobre a estrutura para manter sua propriedade de *heap.* Esta operação é comumente conhecida como *max_heapify* (em *max_heaps*) ou *min_heapify* (em *min_heaps*).

Operações em Heaps

A função *max_heapify* é aplicada sobre o elemento que ocupa a posição *i* no *max_heap h,* de tamanho *n*, para posicioná-lo adequadamente dentro da estrutura, mantendo a propriedade de *heap* *(Cormem et al., 2012)*

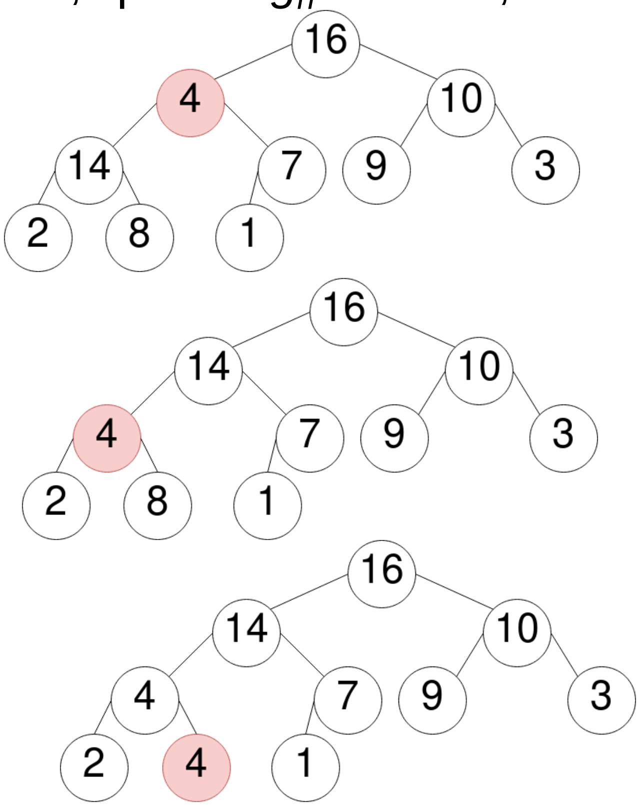
```
max_heapify(h,i,n)
   l = 2i
   r = 2i+1
   se l≤n e h[l]>h[i]
      maior = l
   senão maior = r
   se r≤n e h[r]>h[maior]
      maior = r
   se maior ≠ i
      troca h[maior] com h[i]
      max_heapify(h,maior,n)
```



Operações em Heaps

A função *max_heapify* faz as trocas necessárias e, se for o caso, é chamada recursivamente para tratar do elemento em sua nova posição. O máximo de chamadas recursivas corresponde à altura da árvore, que é $log_n 2$. Assim, a complexidade de *max_heapify* é $O(logn2)$.

```
max_heapify(h,i,n)
   l = 2i
   r = 2i+1
   se l≤n e h[l]>h[i]
      maior = l
   senão maior = r
   se r≤n e h[r]>h[maior]
      maior = r
   se maior ≠ i
      troca h[maior] com h[i]
      max_heapify(h,maior,n)
```



A construção de um Heap

Pode-se transformar uma sequência qualquer em um *heap* trocando os elementos de posição caso necessário para atingir a propriedade *heap.*

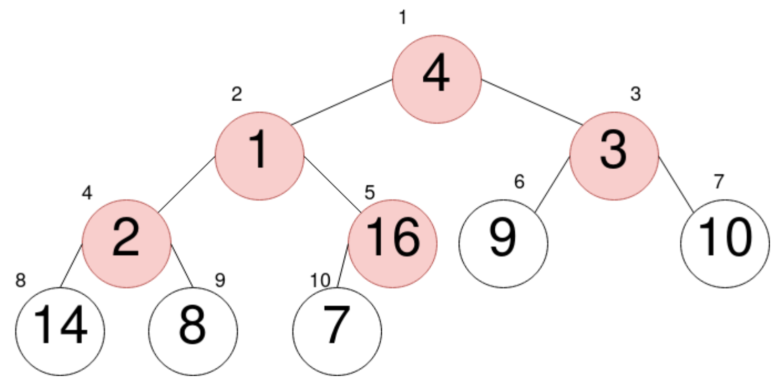
Compara-se cada elemento com seus filhos, trocando-os de posição caso necessário.

Em uma sequência *v* de tamanho *n*, os elementos de v[(⌊(n/2)⌋+1) … n] (i.e. a segunda metade da sequência) são folhas da árvore e não têm com quem serem comparados.

Logo, aplica-se a função *max_heap* à primeira metade da sequência (i.e. v[1 … ⌊(n/2)⌋])

```
build_max_heap(n)
    para i de ⌊tamanho(n)/2⌋ até 1
        max_heapify(n,i)
```

```
     1  2  3  4  5   6  7   8   9  10
v = [ 4, 1, 3, 2, 16, 9, 10, 14, 8, 7 ]
```

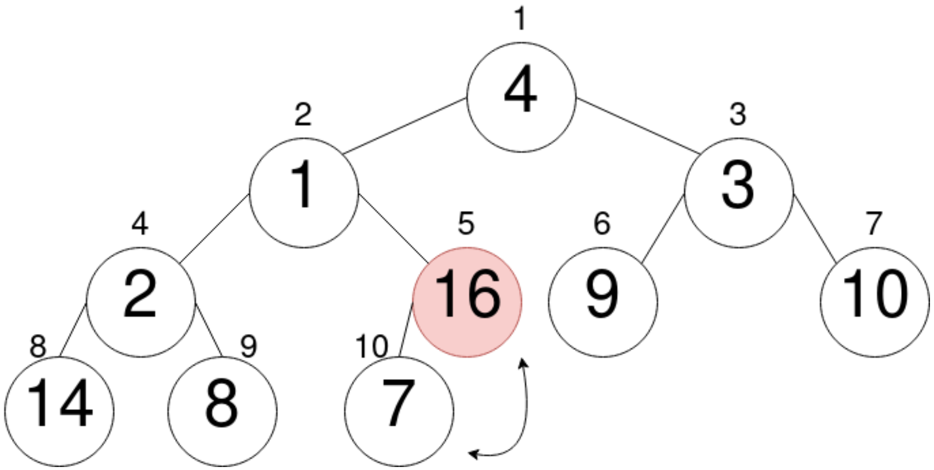


A construção de um Heap

```
build_max_heap(n)
   para i de ⌊tamanho(n)/2⌋ até 1
      max_heapify(n,i)
```

1	2	3	4	5	6	7	8	9	10

v = [4, 1, 3, 2, **16**, 9, 10, 14, 8, **7**]



para *i=5*

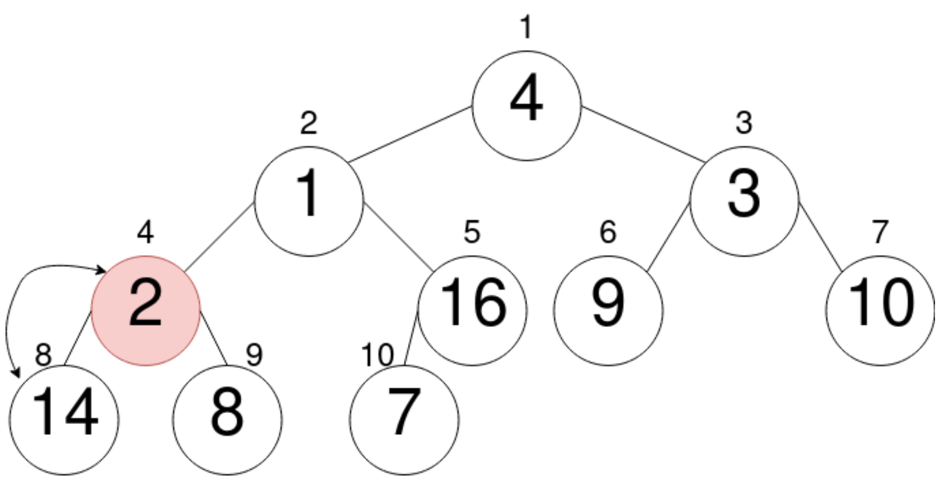
v[i] é maior que seus sucessores logo, nenhuma troca é necessária

A construção de um Heap

```
build_max_heap(n)
   para i de ⌊tamanho(n)/2⌋ até 1
      max_heapify(n,i)
```

1	2	3	4	5	6	7	8	9	10

v = [4, 1, 3, **2**, 16, 9, 10, **14**, 8, 7]



para *i=4*

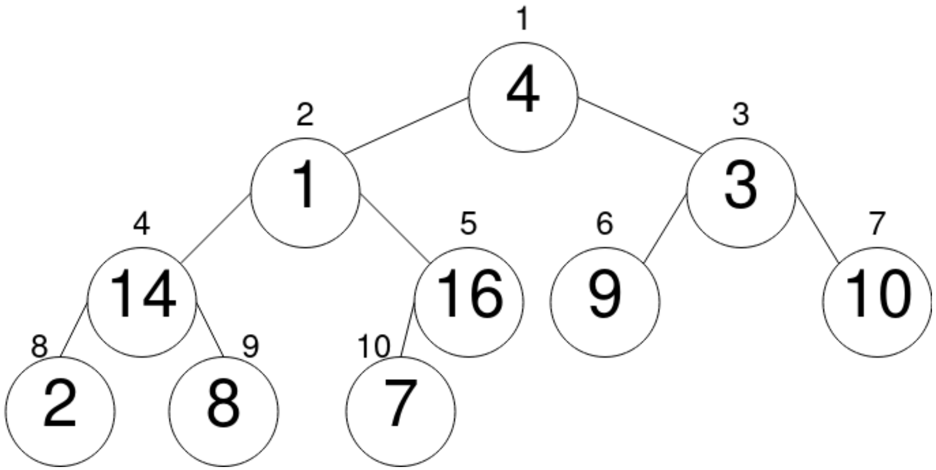
v[i] é menor que seus sucessores

troca-se *v[i]* com o maior de seus sucessores

A construção de um Heap

```
build_max_heap(n)
   para i de ⌊tamanho(n)/2⌋ até 1
      max_heapify(n,i)
```

```
          1    2    3     4     5     6     7     8    9    10
v = [ 4,  1,  3,  14,  16,  9,  10,  2,  8,  7 ]
```



para *i=4*

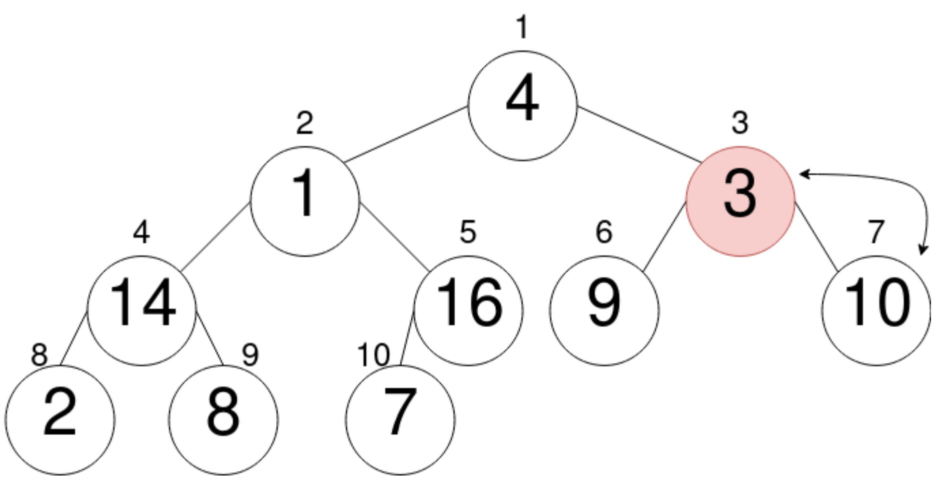
v[i] é menor que seus sucessores

troca-se *v[i]* com o maior de seus sucessores

A construção de um Heap

```
build_max_heap(n)
   para i de ⌊tamanho(n)/2⌋ até 1
      max_heapify(n,i)
```

```
          1   2   3   4    5    6   7    8   9  10
v = [ 4,  1,  3,  14,  16,  9,  10,  2,  8,  7 ]
```



para *i=3*

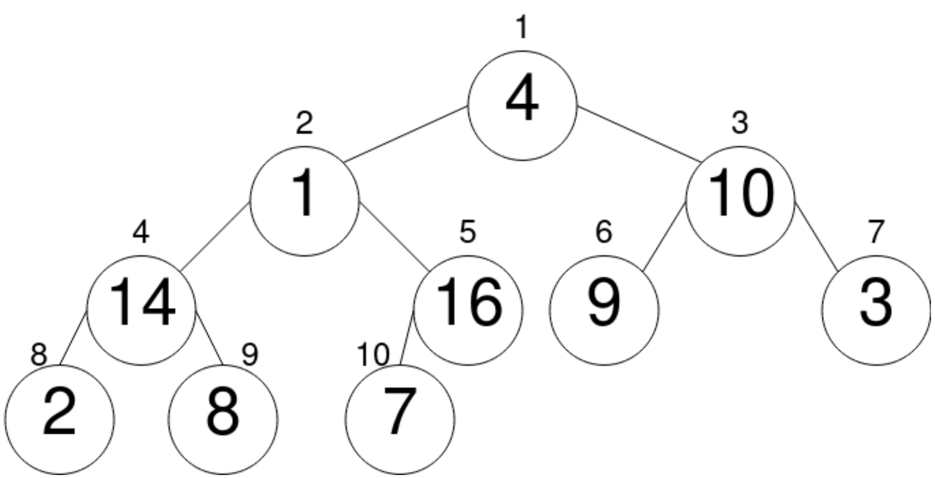
v[i] é menor que seus sucessores

troca-se *v[i]* com o maior de seus sucessores

A construção de um Heap

```
build_max_heap(n)
   para i de ⌊tamanho(n)/2⌋ até 1
      max_heapify(n,i)
```

```
         1   2    3     4     5    6   7   8   9   10
 v = [ 4,  1,  10,  14,  16,  9,  3,  2,  8,  7 ]
```



para *i=3*

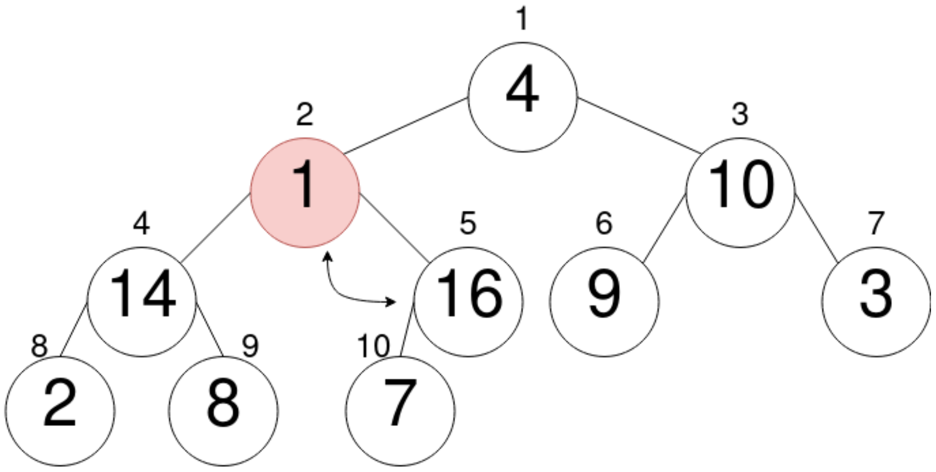
v[i] é menor que seus sucessores

troca-se *v[i]* com o maior de seus sucessores

A construção de um Heap

```
build_max_heap(n)
   para i de ⌊tamanho(n)/2⌋ até 1
      max_heapify(n,i)
```

	1	2	3	4	5	6	7	8	9	10	
v = [	4,	**1**,	10,	14,	**16**,	9,	3,	2,	8,	7	]



para *i=2*

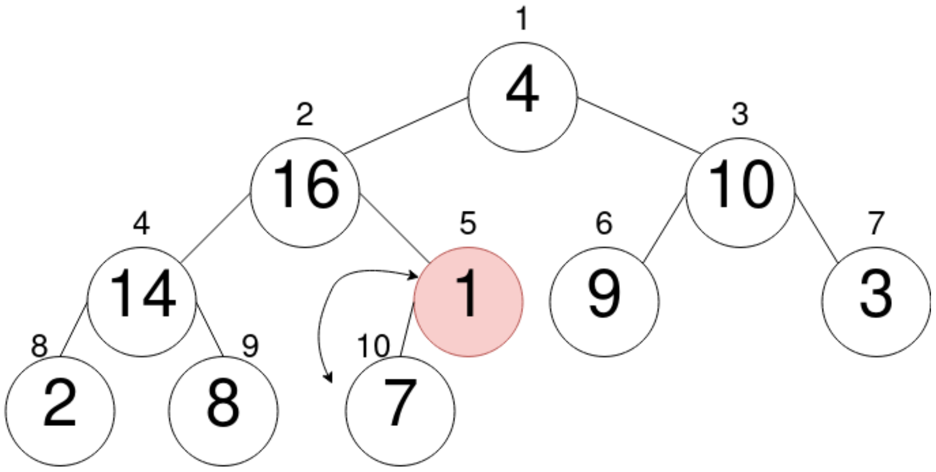
v[i] é menor que seus sucessores

troca-se *v[i]* com o maior de seus sucessores (recursivamente)

A construção de um Heap

```
build_max_heap(n)
   para i de ⌊tamanho(n)/2⌋ até 1
      max_heapify(n,i)
```

	1	2	3	4	5	6	7	8	9	10	
v = [	4,	16,	10,	14,	**1**,	9,	3,	2,	8,	**7**	]



para *i=2*

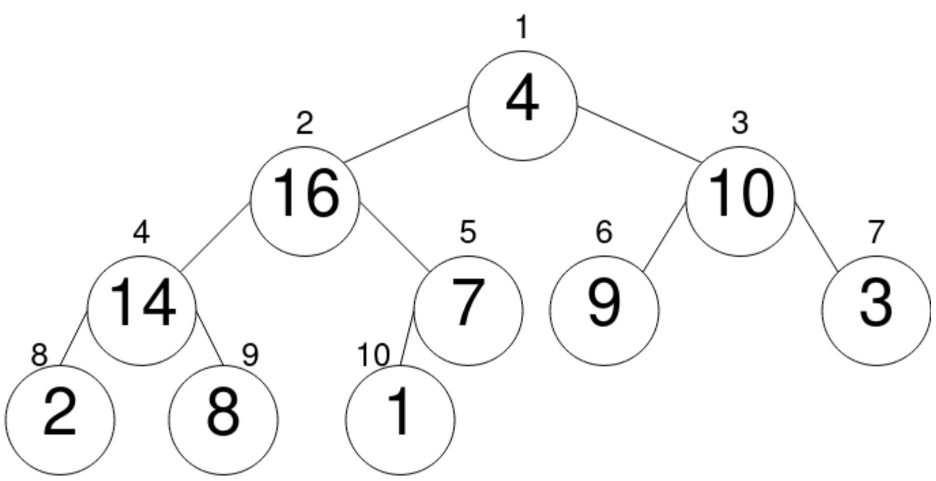
v[i] é menor que seus sucessores

troca-se *v[i]* com o maior de seus sucessores (recursivamente)

A construção de um Heap

```
build_max_heap(n)
   para i de ⌊tamanho(n)/2⌋ até 1
      max_heapify(n,i)
```

	1	2	3	4	5	6	7	8	9	10	
v = [	4,	16,	10,	14,	**1**,	9,	3,	2,	8,	**7**	]



para *i=2*

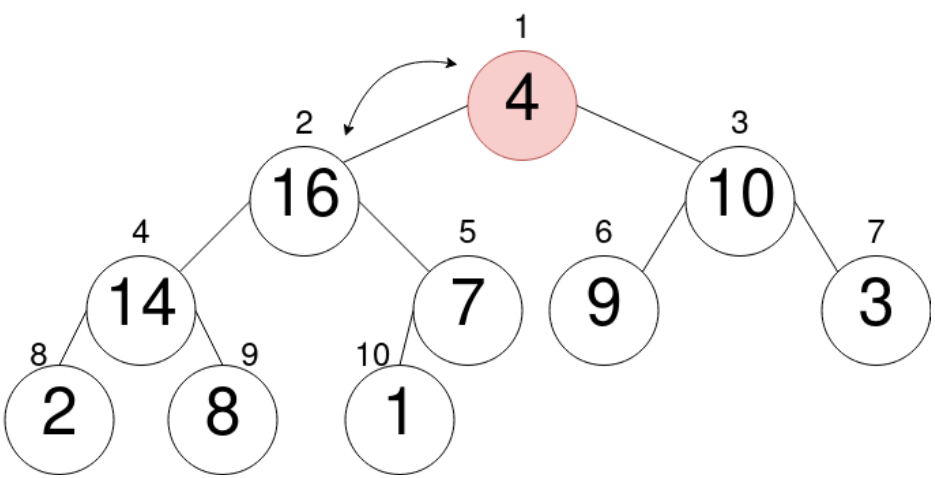
v[i] é menor que seus sucessores

troca-se *v[i]* com o maior de seus sucessores (recursivamente)

A construção de um Heap

```
build_max_heap(n)
   para i de ⌊tamanho(n)/2⌋ até 1
      max_heapify(n,i)
```

```
        1    2     3     4    5    6    7    8    9    10
v = [ 4, 16, 10, 14, 1, 9, 3, 2, 8, 7 ]
```



para *i=1*

v[i] é menor que seus sucessores

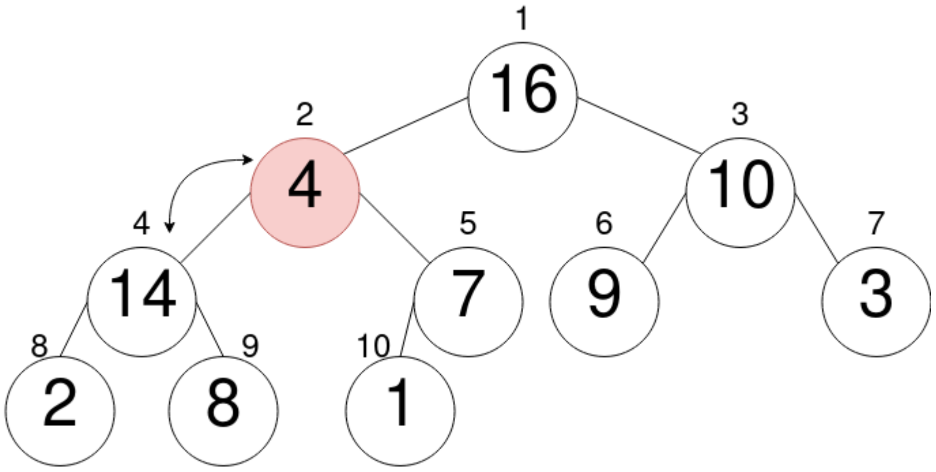
troca-se *v[i]* com o maior de seus sucessores (recursivamente)

A construção de um Heap

```
build_max_heap(n)
   para i de ⌊tamanho(n)/2⌋ até 1
      max_heapify(n,i)
```

1	2	3	4	5	6	7	8	9	10

v = [16, **4**, 10, **14**, 1, 9, 3, 2, 8, 7]



para *i=1*

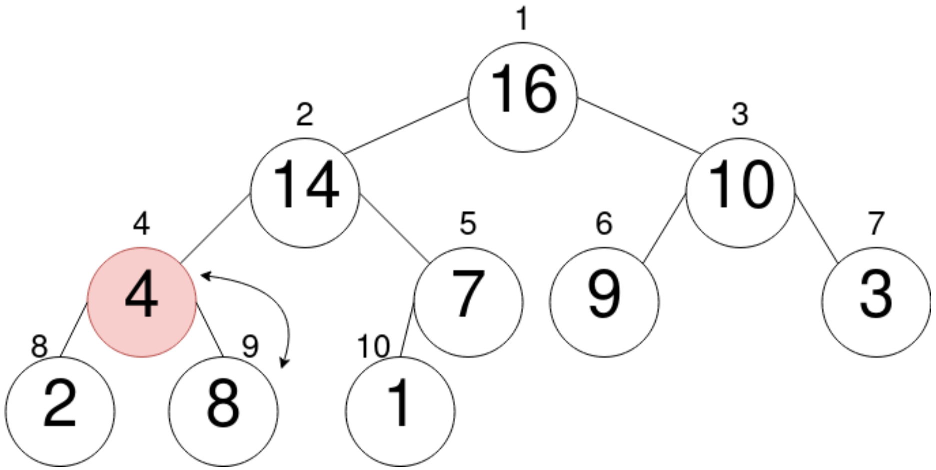
v[i] é menor que seus sucessores

troca-se *v[i]* com o maior de seus sucessores (recursivamente)

A construção de um Heap

```
build_max_heap(n)
   para i de ⌊tamanho(n)/2⌋ até 1
      max_heapify(n,i)
```

```
         1      2     3     4    5     6    7    8    9    10
 v = [ 16, 14, 10, 4, 1, 9, 3, 2, 8, 7 ]
```



para *i=1*

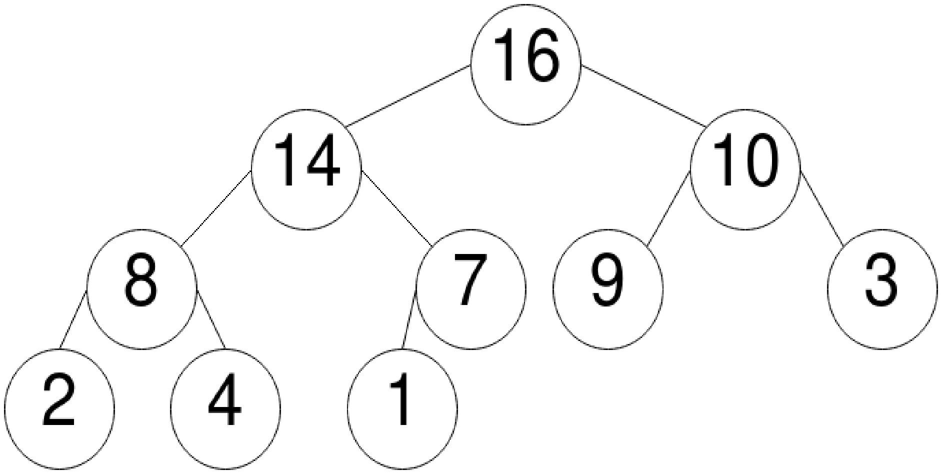
v[i] é menor que seus sucessores

troca-se *v[i]* com o maior de seus sucessores (recursivamente)

A construção de um Heap

```
build_max_heap(n)
   para i de ⌊tamanho(n)/2⌋ até 1
      max_heapify(n,i)
```

```
          1     2    3    4   5    6   7   8   9   10
v = [ 16, 14, 10, 8, 1, 9, 3, 2, 4, 7 ]
```



para *i=1*

v[i] é menor que seus sucessores

troca-se *v[i]* com o maior de seus sucessores (recursivamente)

Filas de prioridades

Uma das aplicações mais populares de heaps são as *filas de prioridades,* que são estruturas em que tem-se acesso unicamente ao elemento de maior prioridade. Essa prioridade é definida por algum valor associado a cada elemento. Se o elemento prioritário é o que tem valor mais alto, utiliza-se um *max-heap.* Se o elemento prioritário é o que tem valor mais baixo, utiliza-se um *min-heap.*

As operações mais comuns em filas de prioridades são:

- *Insert (H,x)*: insere o elemento *x* no heap *H;*
- *Priority(H)*: retorna o elemento prioritário armazenado no heap *H;*
- *Extract(H)*: retorna o elemento prioritário armazenado no heap *H* e o remove da estrutura;
- *UpdateKey(H,i,k)*:altera o valor da chave do *i*-ésimo elemento para *k;*

Filas de prioridades

Priority(H): retorna o elemento prioritário armazenado no heap *H;*

```
priority(H)
   Return H[1]
```

Filas de prioridades

Extract(H): retorna o elemento prioritário armazenado no heap *H* e o remove da estrutura;

```
extract_max(H)
   se tamanho(H)>=1
      max = H[1]
      H[1] = H[tamanho(H)]
      tamanho(H) = tamanho(H)-1
      max_heapify(H, 1, tamanho(H))
```

Filas de prioridades

UpdateKey(H,i,k): altera o valor da chave do *i*-ésimo elemento para *k;*

```
increase_key(H,i,k)
   se k>i
      H[i] = k
      enquanto i>1 e H[⌊i/2⌋] < H[i]
         troca(H[i],H[⌊i/2⌋])
         i = ⌊i/2⌋
```

Filas de prioridades

Insert (H,k): insere o elemento *k* no heap *H;*

```
max_insert(H,k)
   tamanho(H) = tamanho(H) + 1
   H[tamanho(H)) = -1
   increase_key(H,tamanho(H),k)
```

Referências

Kleinberg, Jon M., Tardos, Éva . Algorithm design. Addison-Wesley 2006, ISBN 978-0-321-37291-8, pp. I-XXIII, 1-838

Cormen, Thomas H., Charles E. Leiserson, Ronald L. Rivest and Clifford Stein. Algoritmos: Teoria E Prática. 3 ed.: Campus, 2012. ISBN: 9788535236996.