Disciplina MCTA027-17 - Teoria dos Grafos

TAD Fila de prioridades e Estrutura Heap

Profa. Carla Negri Lintzmayer

carla.negri@ufabc.edu.br

www.professor.ufabc.edu.br/∼carla.negri

Centro de Matemática, Computação e Cognição – Universidade Federal do ABC

Fila de Prioridades

TAD Fila de Prioridades

- Coleção dinâmica de elementos que possuem prioridades associadas e cuja operação de remoção deve remover o elemento que possui maior prioridade.

TAD Fila de Prioridades

- Coleção dinâmica de elementos que possuem prioridades associadas e cuja operação de remoção deve remover o elemento que possui maior prioridade.
- Além da remoção: consulta ao elemento de maior prioridade, inserção de um novo elemento, alteração da prioridade de um elemento já armazenado e construção a partir de um conjunto pré-existente de elementos.

TAD Fila de Prioridades

- Coleção dinâmica de elementos que possuem prioridades associadas e cuja operação de remoção deve remover o elemento que possui maior prioridade.
- Além da remoção: consulta ao elemento de maior prioridade, inserção de um novo elemento, alteração da prioridade de um elemento já armazenado e construção a partir de um conjunto pré-existente de elementos.
- **Prioridade** é usado de maneira "genérica":

TAD Fila de Prioridades

- Coleção dinâmica de elementos que possuem prioridades associadas e cuja operação de remoção deve remover o elemento que possui maior prioridade.
- Além da remoção: consulta ao elemento de maior prioridade, inserção de um novo elemento, alteração da prioridade de um elemento já armazenado e construção a partir de um conjunto pré-existente de elementos.
- **Prioridade** é usado de maneira "genérica":
  - ter maior prioridade não significa necessariamente ter o maior *valor* indicativo de prioridade

TAD Fila de Prioridades

- Coleção dinâmica de elementos que possuem prioridades associadas e cuja operação de remoção deve remover o elemento que possui maior prioridade.
- Além da remoção: consulta ao elemento de maior prioridade, inserção de um novo elemento, alteração da prioridade de um elemento já armazenado e construção a partir de um conjunto pré-existente de elementos.
- **Prioridade** é usado de maneira "genérica":
  - ter maior prioridade não significa necessariamente ter o maior *valor* indicativo de prioridade
  - se prioridade $=$ idade em fila de banco, então 64 tem mais prioridade que 46

TAD Fila de Prioridades

- Coleção dinâmica de elementos que possuem prioridades associadas e cuja operação de remoção deve remover o elemento que possui maior prioridade.
- Além da remoção: consulta ao elemento de maior prioridade, inserção de um novo elemento, alteração da prioridade de um elemento já armazenado e construção a partir de um conjunto pré-existente de elementos.
- **Prioridade** é usado de maneira "genérica":
  - ter maior prioridade não significa necessariamente ter o maior *valor* indicativo de prioridade
  - se prioridade $=$ idade em fila de banco, então 64 tem mais prioridade que 46
  - se prioridade $=$ número de remédios em estoque, então 2 tem mais prioridade que 8 $(-2 > -8)$

TAD Fila de Prioridades

- Seja n o número de elementos armazenados na estrutura.

TAD Fila de Prioridades

- Seja n o número de elementos armazenados na estrutura.
- Pode ser implementada, por exemplo, com um vetor ordenado de forma crescente pelas prioridades:

TAD Fila de Prioridades

- Seja n o número de elementos armazenados na estrutura.
- Pode ser implementada, por exemplo, com um vetor ordenado de forma crescente pelas prioridades:
  - Remoção em $O(1)$ (remoção do último elemento do vetor)

TAD Fila de Prioridades

- Seja n o número de elementos armazenados na estrutura.
- Pode ser implementada, por exemplo, com um vetor ordenado de forma crescente pelas prioridades:
  - Remoção em $O(1)$ (remoção do último elemento do vetor)
  - Inserção em $O(n)$ (movimentar os elementos de $[i+1..n]$)

TAD Fila de Prioridades

- Seja n o número de elementos armazenados na estrutura.
- Pode ser implementada, por exemplo, com um vetor ordenado de forma crescente pelas prioridades:
  - Remoção em $O(1)$ (remoção do último elemento do vetor)
  - Inserção em $O(n)$ (movimentar os elementos de $[i+1..n]$)
  - Consulta em $O(1)$ (acesso ao último elemento)

TAD Fila de Prioridades

- Seja n o número de elementos armazenados na estrutura.
- Pode ser implementada, por exemplo, com um vetor ordenado de forma crescente pelas prioridades:
  - Remoção em $O(1)$ (remoção do último elemento do vetor)
  - Inserção em $O(n)$ (movimentar os elementos de $[i+1..n]$)
  - Consulta em $O(1)$ (acesso ao último elemento)
  - Alteração em $O(n)$ (movimentar os elementos de $[i..j]$)

TAD Fila de Prioridades

- Seja n o número de elementos armazenados na estrutura.
- Pode ser implementada, por exemplo, com um vetor ordenado de forma crescente pelas prioridades:
  - Remoção em $O(1)$ (remoção do último elemento do vetor)
  - Inserção em $O(n)$ (movimentar os elementos de $[i+1..n]$)
  - Consulta em $O(1)$ (acesso ao último elemento)
  - Alteração em $O(n)$ (movimentar os elementos de $[i..j]$)
  - Construção em $O(n \log n)$ (ordenar um vetor já preenchido)

TAD Fila de Prioridades

- Seja n o número de elementos armazenados na estrutura.
- Pode ser implementada, por exemplo, com um vetor ordenado de forma crescente pelas prioridades:
  - Remoção em $O(1)$ (remoção do último elemento do vetor)
  - Inserção em $O(n)$ (movimentar os elementos de $[i+1..n]$)
  - Consulta em $O(1)$ (acesso ao último elemento)
  - Alteração em $O(n)$ (movimentar os elementos de $[i..j]$)
  - Construção em $O(n \log n)$ (ordenar um vetor já preenchido)
- Veremos a estrutura de dados **heap binário**, que permite implementar essas operações de modo eficiente:

TAD Fila de Prioridades

- Seja n o número de elementos armazenados na estrutura.
- Pode ser implementada, por exemplo, com um vetor ordenado de forma crescente pelas prioridades:
  - Remoção em $O(1)$ (remoção do último elemento do vetor)
  - Inserção em $O(n)$ (movimentar os elementos de $[i+1..n]$)
  - Consulta em $O(1)$ (acesso ao último elemento)
  - Alteração em $O(n)$ (movimentar os elementos de $[i..j]$)
  - Construção em $O(n \log n)$ (ordenar um vetor já preenchido)
- Veremos a estrutura de dados **heap binário**, que permite implementar essas operações de modo eficiente:
  - Remoção em $O(\log n)$

TAD Fila de Prioridades

- Seja n o número de elementos armazenados na estrutura.
- Pode ser implementada, por exemplo, com um vetor ordenado de forma crescente pelas prioridades:
  - Remoção em $O(1)$ (remoção do último elemento do vetor)
  - Inserção em $O(n)$ (movimentar os elementos de $[i+1..n]$)
  - Consulta em $O(1)$ (acesso ao último elemento)
  - Alteração em $O(n)$ (movimentar os elementos de $[i..j]$)
  - Construção em $O(n \log n)$ (ordenar um vetor já preenchido)
- Veremos a estrutura de dados **heap binário**, que permite implementar essas operações de modo eficiente:
  - Remoção em $O(\log n)$
  - Inserção em $O(\log n)$

TAD Fila de Prioridades

- Seja n o número de elementos armazenados na estrutura.
- Pode ser implementada, por exemplo, com um vetor ordenado de forma crescente pelas prioridades:
  - Remoção em $O(1)$ (remoção do último elemento do vetor)
  - Inserção em $O(n)$ (movimentar os elementos de $[i+1..n]$)
  - Consulta em $O(1)$ (acesso ao último elemento)
  - Alteração em $O(n)$ (movimentar os elementos de $[i..j]$)
  - Construção em $O(n \log n)$ (ordenar um vetor já preenchido)
- Veremos a estrutura de dados **heap binário**, que permite implementar essas operações de modo eficiente:
  - Remoção em $O(\log n)$
  - Inserção em $O(\log n)$
  - Consulta em $O(1)$

TAD Fila de Prioridades

- Seja n o número de elementos armazenados na estrutura.
- Pode ser implementada, por exemplo, com um vetor ordenado de forma crescente pelas prioridades:
  - Remoção em $O(1)$ (remoção do último elemento do vetor)
  - Inserção em $O(n)$ (movimentar os elementos de $[i+1..n]$)
  - Consulta em $O(1)$ (acesso ao último elemento)
  - Alteração em $O(n)$ (movimentar os elementos de $[i..j]$)
  - Construção em $O(n \log n)$ (ordenar um vetor já preenchido)
- Veremos a estrutura de dados **heap binário**, que permite implementar essas operações de modo eficiente:
  - Remoção em $O(\log n)$
  - Inserção em $O(\log n)$
  - Consulta em $O(1)$
  - Alteração em $O(\log n)$

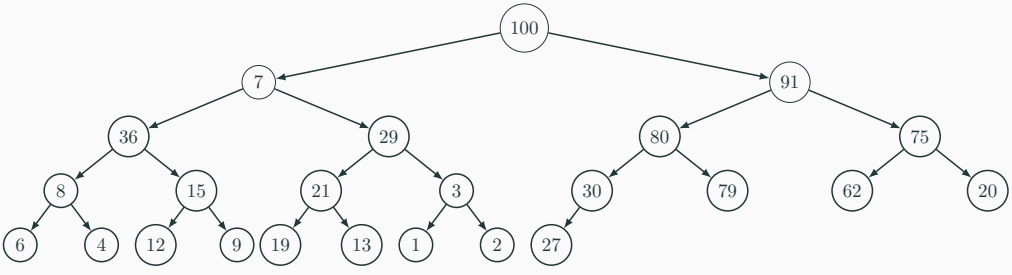
TAD Fila de Prioridades

- Seja n o número de elementos armazenados na estrutura.
- Pode ser implementada, por exemplo, com um vetor ordenado de forma crescente pelas prioridades:
  - Remoção em $O(1)$ (remoção do último elemento do vetor)
  - Inserção em $O(n)$ (movimentar os elementos de $[i+1..n]$)
  - Consulta em $O(1)$ (acesso ao último elemento)
  - Alteração em $O(n)$ (movimentar os elementos de $[i..j]$)
  - Construção em $O(n \log n)$ (ordenar um vetor já preenchido)
- Veremos a estrutura de dados **heap binário**, que permite implementar essas operações de modo eficiente:
  - Remoção em $O(\log n)$
  - Inserção em $O(\log n)$
  - Consulta em $O(1)$
  - Alteração em $O(\log n)$
  - Construção em $O(n)$

Vetores e árvores

Vetores como árvores

É possível visualizar qualquer vetor $A[1..n]$ elementos como uma árvore binária quase completa em que todos os níveis estão cheios, exceto talvez pelo último:



1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23	24
100	7	91	36	29	80	75	8	15	21	3	30	79	62	20	6	4	12	9	19	13	1	2	27

Ao percorrer o vetor da esquerda para a direita, acessamos todos os nós de um nível ℓ consecutivamente antes de acessar os nós do nível $\ell + 1$.

Vetores como árvores

O elemento que está armazenado na posição i do vetor ($1 \leq i \leq n$):

- tem filho esquerdo na posição $2i$, se $2i \leq n$;
- tem filho direito na posição $2i$, se $2i \leq n$;
- tem pai na posição $\lfloor i/2 \rfloor$, se $i > 1$;
- está no nível $\lfloor \lg i \rfloor$;
- tem altura $\lfloor \lg(n/i) \rfloor$;
- pode ser raiz de uma árvore enraizada na posição i;
- pode ser o fim de uma subárvore com os elementos de $A[1..i]$.

Estrutura de dados Heap

Heap

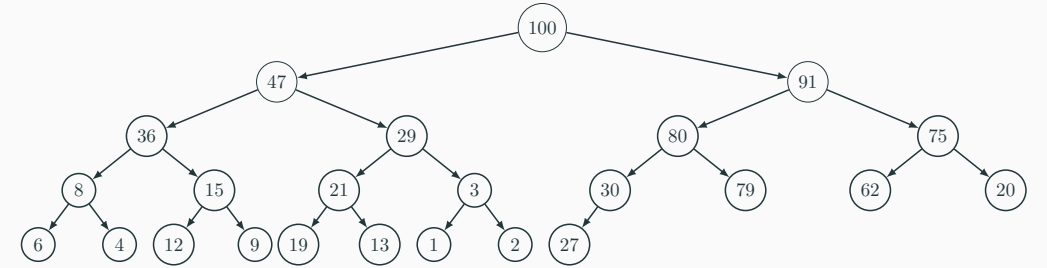
Propriedade de heap

Todo nó tem prioridade maior ou igual à prioridade de seus filhos, se eles existirem.

Heap

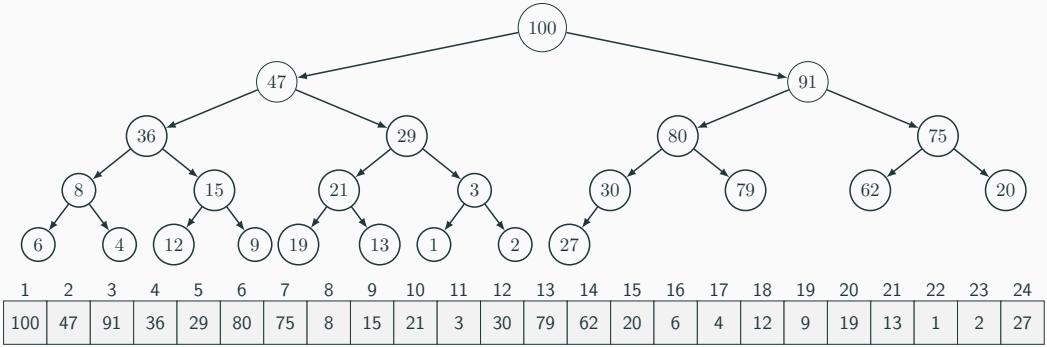
Um vetor A é um Heap se satisfaz a propriedade de heap.

Exemplo de heap



1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23	24
100	47	91	36	29	80	75	8	15	21	3	30	79	62	20	6	4	12	9	19	13	1	2	27

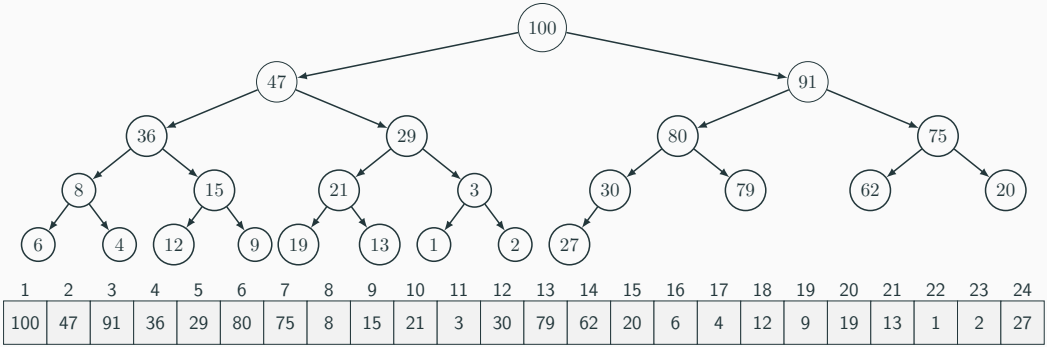
Exemplo de heap



1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23	24
100	47	91	36	29	80	75	8	15	21	3	30	79	62	20	6	4	12	9	19	13	1	2	27

O que podemos dizer sobre $A[1]$?

Exemplo de heap



1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23	24
100	47	91	36	29	80	75	8	15	21	3	30	79	62	20	6	4	12	9	19	13	1	2	27

O que podemos dizer sobre $A[1]$?

E sobre $A[2]$? Ou $A[n]$?

Convenções

- Os rótulos dos elementos são números entre 0 e $n - 1$;
  - Se não forem, podemos usar uma tabela de símbolos para fazer a conversão.

Convenções

- Os rótulos dos elementos são números entre 0 e $n-1$;
  - Se não forem, podemos usar uma tabela de símbolos para fazer a conversão.
- Um elemento e contém rótulo, $e.label$, e uma prioridade, $e.prio$;
  - Poderia ter outras informações...

Convenções

- Os rótulos dos elementos são números entre 0 e $n-1$;
  - Se não forem, podemos usar uma tabela de símbolos para fazer a conversão.
- Um elemento e contém rótulo, $e.label$, e uma prioridade, $e.prio$;
  - Poderia ter outras informações...
- Nosso Heap será um vetor H de elementos;

Convenções

- Os rótulos dos elementos são números entre 0 e $n-1$;
  - Se não forem, podemos usar uma tabela de símbolos para fazer a conversão.
- Um elemento e contém rótulo, $e.label$, e uma prioridade, $e.prio$;
  - Poderia ter outras informações...
- Nosso Heap será um vetor H de elementos;
- Manteremos ainda um vetor ind, que armazenará os índices em H em que os elementos estão armazenados:
  - o elemento de rótulo r está em $H[\, ind[\, r\,]\,]$;

Convenções

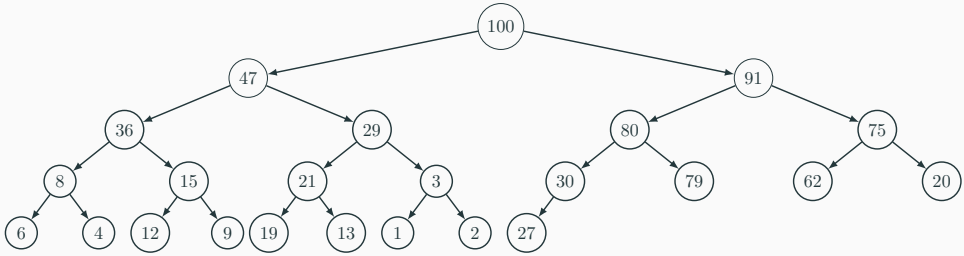
- Os rótulos dos elementos são números entre 0 e $n-1$;
  - Se não forem, podemos usar uma tabela de símbolos para fazer a conversão.
- Um elemento e contém rótulo, $e.label$, e uma prioridade, $e.prio$;
  - Poderia ter outras informações...
- Nosso Heap será um vetor H de elementos;
- Manteremos ainda um vetor ind, que armazenará os índices em H em que os elementos estão armazenados:
  - o elemento de rótulo r está em $H[\,ind[\,r\,]\,]$;
  - a prioridade do elemento de rótulo r é $H[\,ind[\,r\,]\,].prio$;

Convenções

- Os rótulos dos elementos são números entre 0 e $n-1$;
  - Se não forem, podemos usar uma tabela de símbolos para fazer a conversão.
- Um elemento e contém rótulo, $e.label$, e uma prioridade, $e.prio$;
  - Poderia ter outras informações...
- Nosso Heap será um vetor H de elementos;
- Manteremos ainda um vetor ind, que armazenará os índices em H em que os elementos estão armazenados:
  - o elemento de rótulo r está em $H[\, ind[\, r\,]\,]$;
  - a prioridade do elemento de rótulo r é $H[\, ind[\, r\,]\,].prio$;
  - e vale que $H[\, ind[\, r\,]\,].label = r$.

Operações para restaurar a propriedade de Heap

Se já temos um Heap e um único elemento tem sua prioridade **diminuída**, podemos facilmente restaurar a propriedade de Heap:

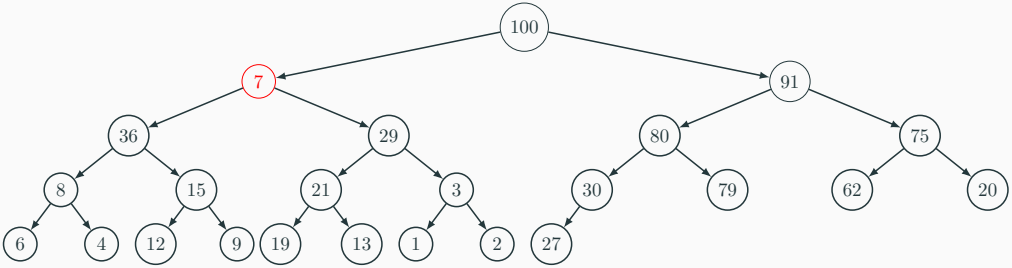


	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23	24
$H.prio$	100	47	91	36	29	80	75	8	15	21	3	30	79	62	20	6	4	12	9	19	13	1	2	27
$H.label$	7	8	16	13	4	1	9	18	5	22	0	14	15	2	20	23	19	12	21	6	3	10	17	11

	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23
ind	11	6	14	21	5	9	20	1	2	7	22	24	18	4	12	13	3	23	8	17	15	19	10	16

Operações para restaurar a propriedade de Heap

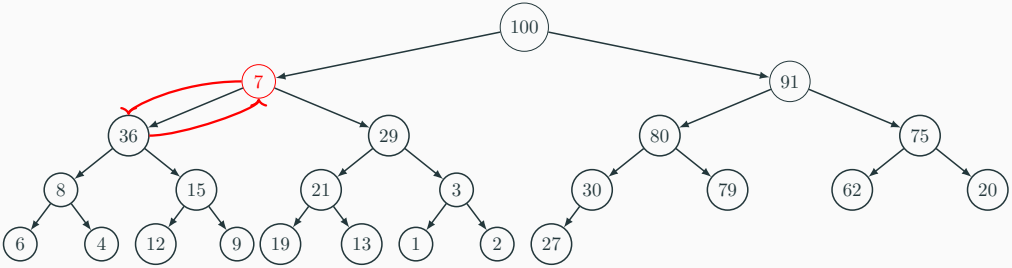
Se já temos um Heap e um único elemento tem sua prioridade **diminuída**, podemos facilmente restaurar a propriedade de Heap:



1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23	24
100	7	91	36	29	80	75	8	15	21	3	30	79	62	20	6	4	12	9	19	13	1	2	27

Operações para restaurar a propriedade de Heap

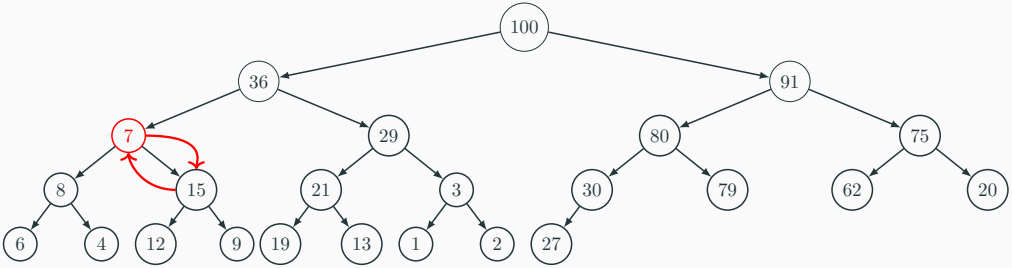
Se já temos um Heap e um único elemento tem sua prioridade **diminuída**, podemos facilmente restaurar a propriedade de Heap:



1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23	24
100	7	91	36	29	80	75	8	15	21	3	30	79	62	20	6	4	12	9	19	13	1	2	27

Operações para restaurar a propriedade de Heap

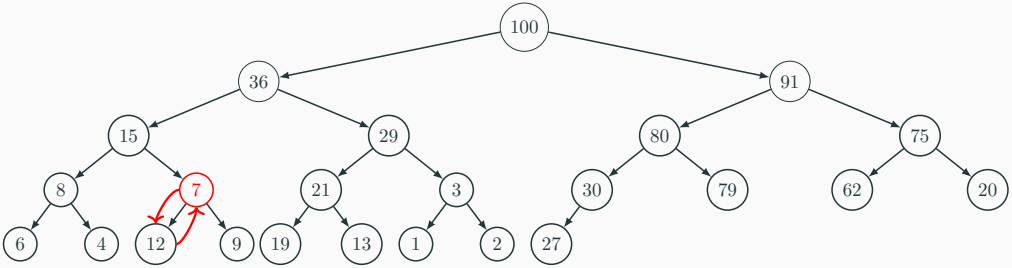
Se já temos um Heap e um único elemento tem sua prioridade **diminuída**, podemos facilmente restaurar a propriedade de Heap:



1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23	24
100	36	91	7	29	80	75	8	15	21	3	30	79	62	20	6	4	12	9	19	13	1	2	27

Operações para restaurar a propriedade de Heap

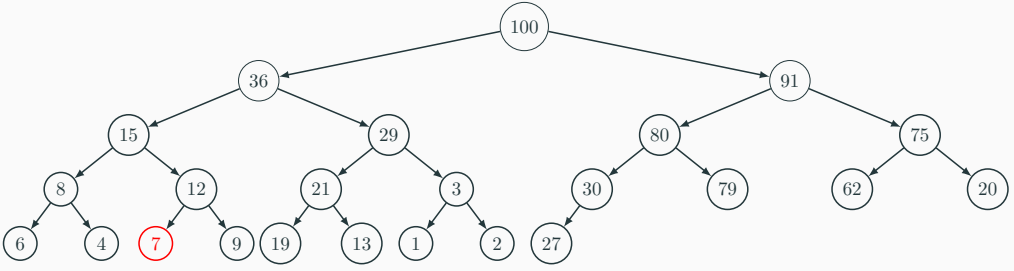
Se já temos um Heap e um único elemento tem sua prioridade **diminuída**, podemos facilmente restaurar a propriedade de Heap:



1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23	24
100	36	91	15	29	80	75	8	7	21	3	30	79	62	20	6	4	12	9	19	13	1	2	27

Operações para restaurar a propriedade de Heap

Se já temos um Heap e um único elemento tem sua prioridade **diminuída**, podemos facilmente restaurar a propriedade de Heap:



1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23	24
100	36	91	15	29	80	75	8	12	21	3	30	79	62	20	6	4	7	9	19	13	1	2	27

Operações para restaurar a propriedade de Heap

```
1:  Função CorrigeHeapDescendo(H, i)
2:      maior ← i
3:      Se 2i ≤ H.tamanho e H[2i].prio > H[maior].prio então
4:          maior ← 2i
5:      Se 2i + 1 ≤ H.tamanho e H[2i + 1].prio > H[maior].prio então
6:          maior ← 2i + 1
7:      Se maior ≠ i então
8:          troca ind[i] com ind[maior]
9:          troca H[i] com H[maior]
10:         CorrigeHeapDescendo(H, maior)
```

Operações para restaurar a propriedade de Heap

```
1:  Função CorrigeHeapDescendo(H, i)
2:      maior ← i
3:      Se 2i ≤ H.tamanho e H[2i].prio > H[maior].prio então
4:          maior ← 2i
5:      Se 2i + 1 ≤ H.tamanho e H[2i + 1].prio > H[maior].prio então
6:          maior ← 2i + 1
7:      Se maior ≠ i então
8:          troca ind[i] com ind[maior]
9:          troca H[i] com H[maior]
10:         CorrigeHeapDescendo(H, maior)
```

Teorema

Seja H um vetor e i um índice tal que as subárvores enraizadas em $H[2i]$ e $H[2i+1]$ são Heaps. Ao final de CorrigeHeapDescendo(H, i), a subárvore enraizada em $H[i]$ é Heap.

Operações para restaurar a propriedade de Heap

```
1:  Função CorrigeHeapDescendo(H, i)
2:      maior ← i
3:      Se 2i ≤ H.tamanho e H[2i].prio > H[maior].prio então
4:          maior ← 2i
5:      Se 2i + 1 ≤ H.tamanho e H[2i + 1].prio > H[maior].prio então
6:          maior ← 2i + 1
7:      Se maior ≠ i então
8:          troca ind[i] com ind[maior]
9:          troca H[i] com H[maior]
10:         CorrigeHeapDescendo(H, maior)
```

Teorema

Seja H um vetor e i um índice tal que as subárvores enraizadas em $H[2i]$ e $H[2i+1]$ são Heaps. Ao final de CorrigeHeapDescendo(H, i), a subárvore enraizada em $H[i]$ é Heap.

A prova é por indução na altura do nó i.

Operações para restaurar a propriedade de Heap

```
1:  Função CorrigeHeapDescendo(H, i)
2:      maior ← i
3:      Se 2i ≤ H.tamanho e H[2i].prio > H[maior].prio então
4:          maior ← 2i
5:      Se 2i + 1 ≤ H.tamanho e H[2i + 1].prio > H[maior].prio então
6:          maior ← 2i + 1
7:      Se maior ≠ i então
8:          troca ind[i] com ind[maior]
9:          troca H[i] com H[maior]
10:         CorrigeHeapDescendo(H, maior)
```

Tempo:

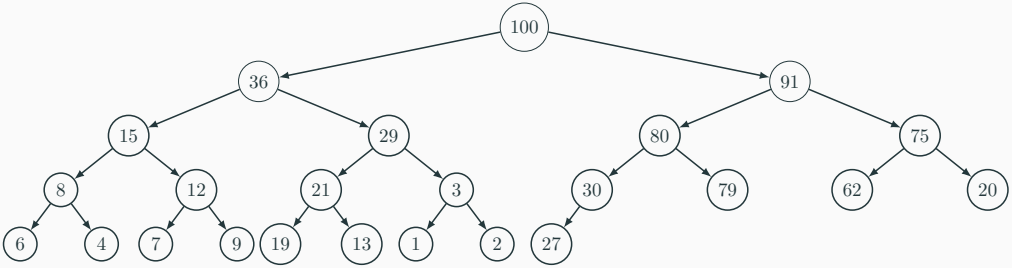
Operações para restaurar a propriedade de Heap

1: **Função** CorrigeHeapDescendo(H, i)
2: $maior \leftarrow i$
3: **Se** $2i \leq H.tamanho$ e $H[2i].prio > H[maior].prio$ **então**
4: $maior \leftarrow 2i$
5: **Se** $2i + 1 \leq H.tamanho$ e $H[2i + 1].prio > H[maior].prio$ **então**
6: $maior \leftarrow 2i + 1$
7: **Se** $maior \neq i$ **então**
8: troca $ind[i]$ com $ind[maior]$
9: troca $H[i]$ com $H[maior]$
10: CorrigeHeapDescendo(H, $maior$)

Tempo: $O(\lg n)$, pois percorremos no máximo um único ramo da árvore, indo de um nó até uma folha, fazendo apenas operações constantes. Como esse ramo contém h nós, onde h é a altura do nó, e a árvore tem altura $\lg n$, o resultado segue.

Operações para restaurar a propriedade de Heap

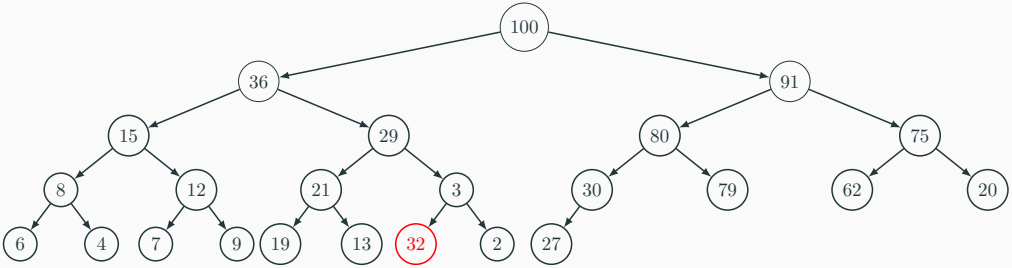
Se já temos um Heap e um único elemento tem sua prioridade **aumentada**, podemos facilmente restaurar a propriedade de Heap:



1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23	24
100	36	91	15	29	80	75	8	12	21	3	30	79	62	20	6	4	7	9	19	13	32	2	27

Operações para restaurar a propriedade de Heap

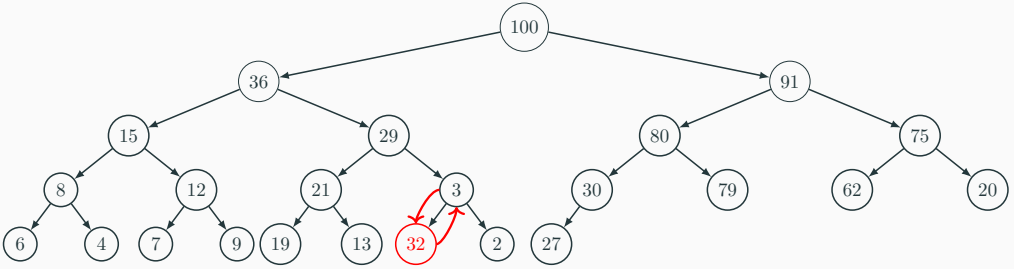
Se já temos um Heap e um único elemento tem sua prioridade **aumentada**, podemos facilmente restaurar a propriedade de Heap:



1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23	24
100	36	91	15	29	80	75	8	12	21	3	30	79	62	20	6	4	7	9	19	13	32	2	27

Operações para restaurar a propriedade de Heap

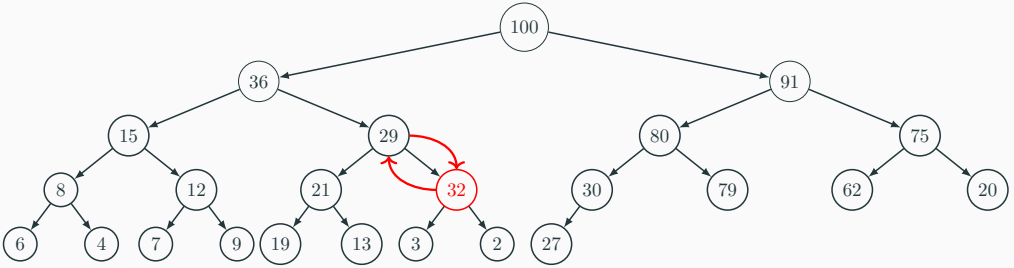
Se já temos um Heap e um único elemento tem sua prioridade **aumentada**, podemos facilmente restaurar a propriedade de Heap:



1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23	24
100	36	91	15	29	80	75	8	12	21	3	30	79	62	20	6	4	7	9	19	13	32	2	27

Operações para restaurar a propriedade de Heap

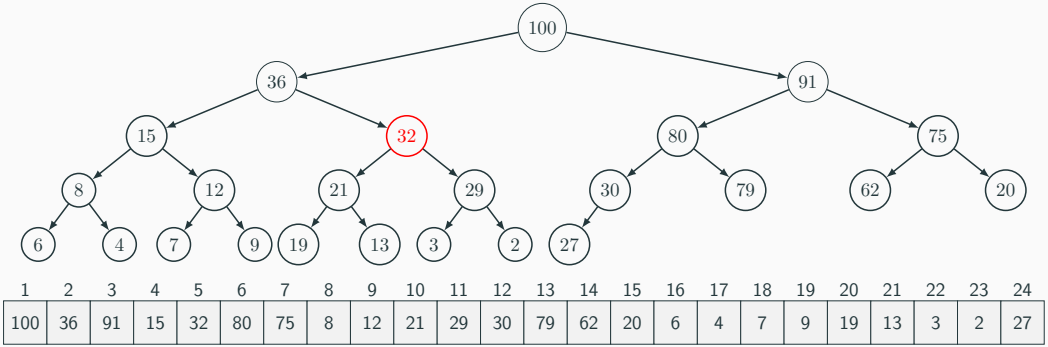
Se já temos um Heap e um único elemento tem sua prioridade **aumentada**, podemos facilmente restaurar a propriedade de Heap:



1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23	24
100	36	91	15	29	80	75	8	12	21	32	30	79	62	20	6	4	7	9	19	13	3	2	27

Operações para restaurar a propriedade de Heap

Se já temos um Heap e um único elemento tem sua prioridade **aumentada**, podemos facilmente restaurar a propriedade de Heap:



1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23	24
100	36	91	15	32	80	75	8	12	21	29	30	79	62	20	6	4	7	9	19	13	3	2	27

Operações para restaurar a propriedade de Heap

```
1: Função CorrigeHeapSubindo(H, i)
2:     pai ← ⌊i/2⌋
3:     Se i ≥ 2 e H[i].prio > P[pai].prio então
4:         troca ind[i] com ind[pai]
5:         troca H[i] com H[pai]
6:         CorrigeHeapSubindo(H, pai)
```

Operações para restaurar a propriedade de Heap

```
1: Função CorrigeHeapSubindo(H, i)
2:     pai ← ⌊i/2⌋
3:     Se i ≥ 2 e H[i].prio > P[pai].prio então
4:         troca ind[i] com ind[pai]
5:         troca H[i] com H[pai]
6:         CorrigeHeapSubindo(H, pai)
```

Teorema

Seja H um vetor e i um índice tal que as subárvores $H[1..i-1]$ é Heap. Ao final de CorrigeHeapSubindo(H, i), a subárvore $H[1..i]$ é Heap.

Operações para restaurar a propriedade de Heap

```
1: Função CorrigeHeapSubindo(H, i)
2:     pai ← ⌊i/2⌋
3:     Se i ≥ 2 e H[i].prio > P[pai].prio então
4:         troca ind[i] com ind[pai]
5:         troca H[i] com H[pai]
6:         CorrigeHeapSubindo(H, pai)
```

Teorema

Seja H um vetor e i um índice tal que as subárvores $H[1..i-1]$ é Heap. Ao final de CorrigeHeapSubindo(H, i), a subárvore $H[1..i]$ é Heap.

Prova por indução no nível do nó i.

Operações para restaurar a propriedade de Heap

```
1: Função CorrigeHeapSubindo(H, i)
2:     pai ← ⌊i/2⌋
3:     Se i ≥ 2 e H[i].prio > P[pai].prio então
4:         troca ind[i] com ind[pai]
5:         troca H[i] com H[pai]
6:         CorrigeHeapSubindo(H, pai)
```

Tempo:

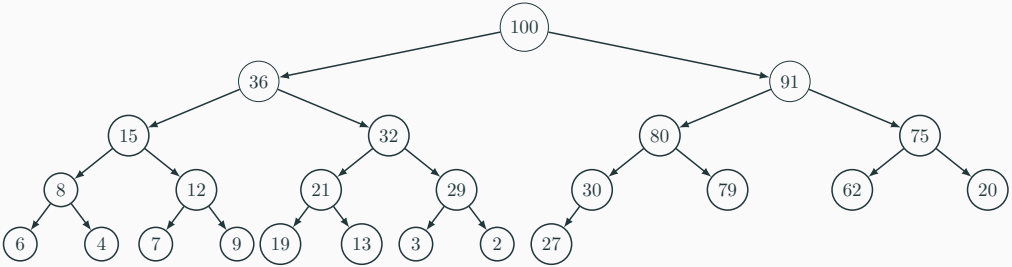
Operações para restaurar a propriedade de Heap

```
1: Função CorrigeHeapSubindo(H, i)
2:     pai ← ⌊i/2⌋
3:     Se i ≥ 2 e H[i].prio > P[pai].prio então
4:         troca ind[i] com ind[pai]
5:         troca H[i] com H[pai]
6:         CorrigeHeapSubindo(H, pai)
```

Tempo: $O(\lg n)$, pois percorremos no máximo um único ramo da árvore, indo de um nó até a raiz, fazendo apenas operações constantes. Como esse ramo contém ℓ nós, onde ℓ é o nível do nó inicial, e o maior nível de um nó é $\lg n$, o resultado segue.

Inserção em Heap

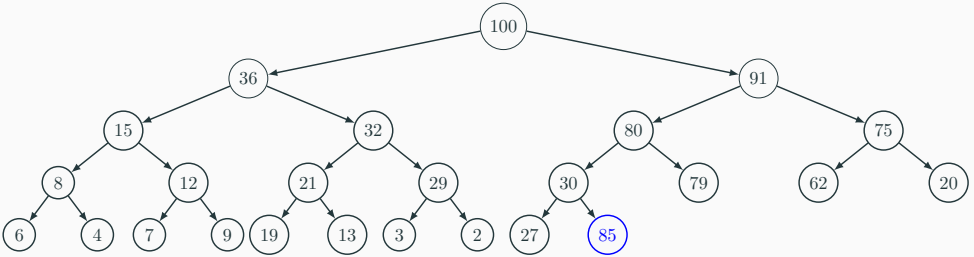
Dado que já temos um Heap, uma inserção ao final do vetor é o que "estraga menos" a nossa propriedade.



1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23	24
100	36	91	15	32	80	75	8	12	21	29	30	79	62	20	6	4	7	9	19	13	3	2	27

Inserção em Heap

Dado que já temos um Heap, uma inserção ao final do vetor é o que "estraga menos" a nossa propriedade.



1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23	24	25
100	36	91	15	32	80	75	8	12	21	29	30	79	62	20	6	4	7	9	19	13	3	2	27	85

Inserção em Heap

Dado que já temos um Heap, uma inserção ao final do vetor é o que "estraga menos" a nossa propriedade.

1: **Função** InsereNaHeap(H, r, p)
2: **Se** $H.tamanho < H.capacidade$ **então**
3: $H.tamanho \leftarrow H.tamanho + 1$
4: $ind[r] \leftarrow H.tamanho$
5: $H[H.tamanho].label \leftarrow r$
6: $H[H.tamanho].prio \leftarrow p$
7: CorrigeHeapSubindo(H, $H.tamanho$)

Tempo:

Inserção em Heap

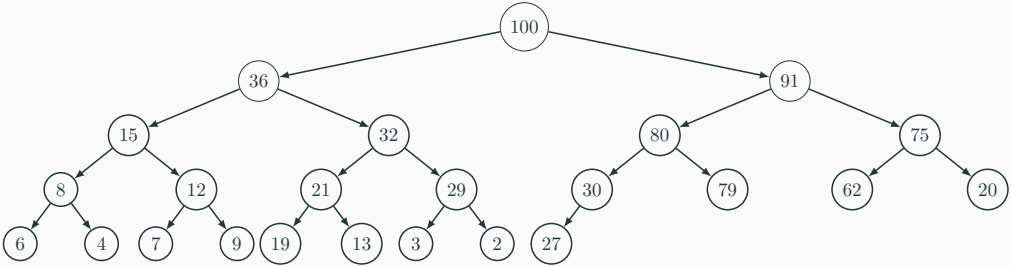
Dado que já temos um Heap, uma inserção ao final do vetor é o que "estraga menos" a nossa propriedade.

1: **Função** InsereNaHeap(H, r, p)
2: **Se** $H.tamanho < H.capacidade$ **então**
3: $H.tamanho \leftarrow H.tamanho + 1$
4: $ind[r] \leftarrow H.tamanho$
5: $H[H.tamanho].label \leftarrow r$
6: $H[H.tamanho].prio \leftarrow p$
7: CorrigeHeapSubindo(H, $H.tamanho$)

Tempo: $O(\log n)$

Remoção em Heap

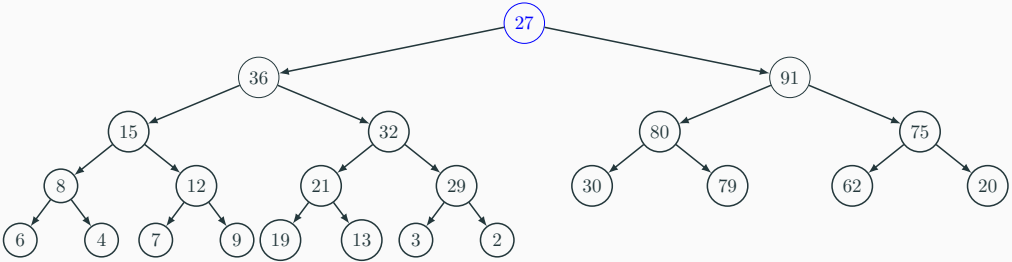
A remoção deve ser pelo elemento de maior prioridade.



1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23	24
100	36	91	15	32	80	75	8	12	21	29	30	79	62	20	6	4	7	9	19	13	3	2	27

Remoção em Heap

A remoção deve ser pelo elemento de maior prioridade.



1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23
27	36	91	15	32	80	75	8	12	21	29	30	79	62	20	6	4	7	9	19	13	3	2

Remoção em Heap

A remoção deve ser pelo elemento de maior prioridade.

1: **Função** RemoveNaHeap(H)
2: $e \leftarrow Null$
3: **Se** $H.tamanho \geq 1$ **então**
4: $e = H[1]$
5: $ind[H[P.tamanho].label] \leftarrow 1$
6: $H[1] \leftarrow H[P.tamanho]$
7: $H.tamanho \leftarrow H.tamanho - 1$
8: CorrigeHeapDescendo(H, 1)
9: **Devolve** e

Tempo:

Remoção em Heap

A remoção deve ser pelo elemento de maior prioridade.

1: **Função** RemoveNaHeap(H)
2: $e \leftarrow Null$
3: **Se** $H.tamanho \geq 1$ **então**
4: $e = H[1]$
5: $ind[H[P.tamanho].label] \leftarrow 1$
6: $H[1] \leftarrow H[P.tamanho]$
7: $H.tamanho \leftarrow H.tamanho - 1$
8: CorrigeHeapDescendo(H, 1)
9: **Devolve** e

Tempo: $O(\log n)$

Alteração em Heap

Um elemento pode ter sua prioridade diminuída ou aumentada.

1: **Função** AlteraHeap(H, r, p)
2: $i \leftarrow ind[r]$
3: **Se** $H[i].prio > p$ **então**
4: $H[i].prio \leftarrow p$
5: CorrigeHeapDescendo(H, i)
6: **Senão Se** $H[i].prio < p$ **então**
7: $H[i].prio \leftarrow p$
8: CorrigeHeapSubindo(H, i)

Tempo:

Alteração em Heap

Um elemento pode ter sua prioridade diminuída ou aumentada.

```
1: Função AlteraHeap(H, r, p)
2:     i ← ind[r]
3:     Se H[i].prio > p então
4:         H[i].prio ← p
5:         CorrigeHeapDescendo(H, i)
6:     Senão Se H[i].prio < p então
7:         H[i].prio ← p
8:         CorrigeHeapSubindo(H, i)
```

Tempo: $O(\log n)$

Construção de Heap

Podemos transformar um vetor já preenchido de forma que ele satisfaça a propriedade de Heap.

1: **Função** ConstroiHeap(H, n)
2: $H.tamanho \leftarrow n$
3: **Para** $i \leftarrow 1$ até n **faça**
4: $ind[H[i].label] = i$
5: **Para** $i \leftarrow \lfloor n/2 \rfloor$ até 1 **faça**
6: CorrigeHeapDescendo(H, i)

Tempo:

Construção de Heap

Podemos transformar um vetor já preenchido de forma que ele satisfaça a propriedade de Heap.

1: **Função** ConstroiHeap(H, n)
2: $H.tamanho \leftarrow n$
3: **Para** $i \leftarrow 1$ até n **faça**
4: $ind[H[i].label] = i$
5: **Para** $i \leftarrow \lfloor n/2 \rfloor$ até 1 **faça**
6: CorrigeHeapDescendo(H, i)

Tempo: $O(n)$

Spoiler

Código i

```
1  // priority_queue.h
2
3  #ifndef __PRIORITY_QUEUE_H_
4  #define __PRIORITY_QUEUE_H_
5
6  typedef struct priority_queue* PQ;
7
8  PQ pq(int);
9  void pq_destroy(PQ);
10
11 int pq_size(PQ);
12
13 void pq_insert(PQ, int label, double prio);
14 int pq_contains(PQ, int label);
15 int pq_extract(PQ);
16 void pq_change(PQ, int label, double new_prio);
17
18 #endif // __PRIORITY_QUEUE_H_
```

Código ii

```c
// heap.c
#include "priority_queue.h"

typedef struct {
    int label;
    double prio;
} Elem;

struct priority_queue {
    Elem* queue;
    int* ind;
    int size;
    int capacity;
};

int parent(int ind) {
    return floor(ind / 2.0);
}

int left_children(int ind) {
    return 2 * ind;
```

Código iii

```
22  }
23
24  int right_children(int ind) {
25      return 2 * ind + 1;
26  }
27
28  PQ pq(int max_size) {
29      PQ pq = Malloc(sizeof(*pq));
30      pq->size = 0;
31      pq->capacity = max_size;
32      pq->queue = Malloc((max_size + 1) * sizeof(*pq->queue)); // use indices 1..max_size
33      pq->ind = Malloc(max_size * sizeof(*pq->ind));
34      for (int i = 0; i < max_size; i++)
35          pq->ind[i] = -1;
36      return pq;
37  }
38
```