Listas de Prioridades – Heap

Prof. Carlos Vinicius G. C. Lima
(vinicius.lima@ufca.edu.br)

Algoritmos e Estruturas de Dados II

Universidade Federal do Cariri – UFCA
(2023/01)

Sumário

Prioridades

Prioridades

- Em muitos problemas se exige a determinação de uma ordem de prioridades aos seus dados.

Prioridades

- Em muitos problemas se exige a determinação de uma ordem de prioridades aos seus dados.
- A **prioridade** é uma informação que revela a importância do dado.

Prioridades

- Em muitos problemas se exige a determinação de uma ordem de prioridades aos seus dados.
- A **prioridade** é uma informação que revela a importância do dado.
  - **Ex:** Priorizar as disciplinas do curso.

Prioridades

- Em muitos problemas se exige a determinação de uma ordem de prioridades aos seus dados.
- A **prioridade** é uma informação que revela a importância do dado.
  - **Ex:** Priorizar as disciplinas do curso.
  - As prioridades podem ser alteradas.

Prioridades

- Em muitos problemas se exige a determinação de uma ordem de prioridades aos seus dados.
- A **prioridade** é uma informação que revela a importância do dado.
  - **Ex:** Priorizar as disciplinas do curso.
  - As prioridades podem ser alteradas.
  - Novas disciplinas podem ser adicionadas aos dados, assim como suas prioridades.

Prioridades

- Em muitos problemas se exige a determinação de uma ordem de prioridades aos seus dados.
- A **prioridade** é uma informação que revela a importância do dado.
  - **Ex:** Priorizar as disciplinas do curso.
  - As prioridades podem ser alteradas.
  - Novas disciplinas podem ser adicionadas aos dados, assim como suas prioridades.
- Objetivo: Organizar os dados de modo a determinar de forma **rápida** o dado de maior prioridade.

Prioridades

- Em muitos problemas se exige a determinação de uma ordem de prioridades aos seus dados.
- A **prioridade** é uma informação que revela a importância do dado.
  - **Ex:** Priorizar as disciplinas do curso.
  - As prioridades podem ser alteradas.
  - Novas disciplinas podem ser adicionadas aos dados, assim como suas prioridades.
- Objetivo: Organizar os dados de modo a determinar de forma **rápida** o dado de maior prioridade.
- Operações desejadas:

Prioridades

- Em muitos problemas se exige a determinação de uma ordem de prioridades aos seus dados.
- A **prioridade** é uma informação que revela a importância do dado.
  - **Ex:** Priorizar as disciplinas do curso.
  - As prioridades podem ser alteradas.
  - Novas disciplinas podem ser adicionadas aos dados, assim como suas prioridades.
- Objetivo: Organizar os dados de modo a determinar de forma **rápida** o dado de maior prioridade.
- Operações desejadas:
  - **Buscar** (sempre) o dado de maior prioridade.

Prioridades

- Em muitos problemas se exige a determinação de uma ordem de prioridades aos seus dados.
- A **prioridade** é uma informação que revela a importância do dado.
  - **Ex:** Priorizar as disciplinas do curso.
  - As prioridades podem ser alteradas.
  - Novas disciplinas podem ser adicionadas aos dados, assim como suas prioridades.
- Objetivo: Organizar os dados de modo a determinar de forma **rápida** o dado de maior prioridade.
- Operações desejadas:
  - **Buscar** (sempre) o dado de maior prioridade.
  - **Remover** (sempre) o dado de maior prioridade.

Prioridades

- Em muitos problemas se exige a determinação de uma ordem de prioridades aos seus dados.
- A **prioridade** é uma informação que revela a importância do dado.
  - **Ex:** Priorizar as disciplinas do curso.
  - As prioridades podem ser alteradas.
  - Novas disciplinas podem ser adicionadas aos dados, assim como suas prioridades.
- Objetivo: Organizar os dados de modo a determinar de forma **rápida** o dado de maior prioridade.
- Operações desejadas:
  - **Buscar** (sempre) o dado de maior prioridade.
  - **Remover** (sempre) o dado de maior prioridade.
  - **Adicionar** um novo dado.

Prioridades

- Em muitos problemas se exige a determinação de uma ordem de prioridades aos seus dados.
- A **prioridade** é uma informação que revela a importância do dado.
  - **Ex:** Priorizar as disciplinas do curso.
  - As prioridades podem ser alteradas.
  - Novas disciplinas podem ser adicionadas aos dados, assim como suas prioridades.
- Objetivo: Organizar os dados de modo a determinar de forma **rápida** o dado de maior prioridade.
- Operações desejadas:
  - **Buscar** (sempre) o dado de maior prioridade.
  - **Remover** (sempre) o dado de maior prioridade.
  - **Adicionar** um novo dado.
  - **Alteração** da prioridade de um dado (tanto aumentando quanto diminuindo).

Métodos

- Pode-se usar diferentes métodos para alcançar o objetivo.

Métodos

- Pode-se usar diferentes métodos para alcançar o objetivo.
  - Lista não ordenada.

Métodos

- Pode-se usar diferentes métodos para alcançar o objetivo.
  - Lista não ordenada.
  - Lista ordenada.

Métodos

- Pode-se usar diferentes métodos para alcançar o objetivo.
  - Lista não ordenada.
  - Lista ordenada.
  - **Heap**.

Métodos

- Pode-se usar diferentes métodos para alcançar o objetivo.
  - Lista não ordenada.
  - Lista ordenada.
  - **Heap**.
- Comparamos a complexidade das operações desejadas para escolher o melhor método.

Listas Ordenadas e Não ordenadas

Listas Não Ordenadas: Listamos os dados em uma lista de tamanho n.

Listas Ordenadas e Não ordenadas

Listas Não Ordenadas: Listamos os dados em uma lista de tamanho n.

Listas Ordenadas: Ordenamos em uma lista de tamanho n os dados de acordo com suas prioridades em ordem decrescente.

Listas Ordenadas e Não ordenadas

Listas Não Ordenadas: Listamos os dados em uma lista de tamanho n.

Listas Ordenadas: Ordenamos em uma lista de tamanho n os dados de acordo com suas prioridades em ordem decrescente.

	Lista Não Ordenadas	Lista Ordenada
Seleção		
Inserção		
Remoção		
Alteração		
Construção		

Listas Ordenadas e Não ordenadas

Listas Não Ordenadas: Listamos os dados em uma lista de tamanho n.

Listas Ordenadas: Ordenamos em uma lista de tamanho n os dados de acordo com suas prioridades em ordem decrescente.

	Lista Não Ordenadas	Lista Ordenada
Seleção	$O(n)$	
Inserção		
Remoção		
Alteração		
Construção		

Veremos um método mais eficiente chamado **heap**.

Listas Ordenadas e Não ordenadas

Listas Não Ordenadas: Listamos os dados em uma lista de tamanho n.

Listas Ordenadas: Ordenamos em uma lista de tamanho n os dados de acordo com suas prioridades em ordem decrescente.

	Lista Não Ordenadas	Lista Ordenada
Seleção	$O(n)$	$O(1)$
Inserção		
Remoção		
Alteração		
Construção		

Veremos um método mais eficiente chamado **heap**.

Listas Ordenadas e Não ordenadas

Listas Não Ordenadas: Listamos os dados em uma lista de tamanho n.

Listas Ordenadas: Ordenamos em uma lista de tamanho n os dados de acordo com suas prioridades em ordem decrescente.

	Lista Não Ordenadas	Lista Ordenada
Seleção	$O(n)$	$O(1)$
Inserção	$O(1)$	
Remoção		
Alteração		
Construção		

Veremos um método mais eficiente chamado **heap**.

Listas Ordenadas e Não ordenadas

Listas Não Ordenadas: Listamos os dados em uma lista de tamanho n.

Listas Ordenadas: Ordenamos em uma lista de tamanho n os dados de acordo com suas prioridades em ordem decrescente.

	Lista Não Ordenadas	Lista Ordenada
Seleção	$O(n)$	$O(1)$
Inserção	$O(1)$	$O(n)$
Remoção		
Alteração		
Construção		

Veremos um método mais eficiente chamado **heap**.

Listas Ordenadas e Não ordenadas

Listas Não Ordenadas: Listamos os dados em uma lista de tamanho n.

Listas Ordenadas: Ordenamos em uma lista de tamanho n os dados de acordo com suas prioridades em ordem decrescente.

	Lista Não Ordenadas	Lista Ordenada
Seleção	$O(n)$	$O(1)$
Inserção	$O(1)$	$O(n)$
Remoção	$O(n)$	
Alteração		
Construção		

Veremos um método mais eficiente chamado **heap**.

Listas Ordenadas e Não ordenadas

Listas Não Ordenadas: Listamos os dados em uma lista de tamanho n.

Listas Ordenadas: Ordenamos em uma lista de tamanho n os dados de acordo com suas prioridades em ordem decrescente.

	Lista Não Ordenadas	Lista Ordenada
Seleção	$O(n)$	$O(1)$
Inserção	$O(1)$	$O(n)$
Remoção	$O(n)$	$O(1)$
Alteração		
Construção		

Veremos um método mais eficiente chamado **heap**.

Listas Ordenadas e Não ordenadas

Listas Não Ordenadas: Listamos os dados em uma lista de tamanho n.

Listas Ordenadas: Ordenamos em uma lista de tamanho n os dados de acordo com suas prioridades em ordem decrescente.

	Lista Não Ordenadas	Lista Ordenada
Seleção	$O(n)$	$O(1)$
Inserção	$O(1)$	$O(n)$
Remoção	$O(n)$	$O(1)$
Alteração	$O(n)$	
Construção		

Veremos um método mais eficiente chamado **heap**.

Listas Ordenadas e Não ordenadas

Listas Não Ordenadas: Listamos os dados em uma lista de tamanho n.

Listas Ordenadas: Ordenamos em uma lista de tamanho n os dados de acordo com suas prioridades em ordem decrescente.

	Lista Não Ordenadas	Lista Ordenada
Seleção	$O(n)$	$O(1)$
Inserção	$O(1)$	$O(n)$
Remoção	$O(n)$	$O(1)$
Alteração	$O(n)$	$O(n)$
Construção		

Veremos um método mais eficiente chamado **heap**.

Listas Ordenadas e Não ordenadas

Listas Não Ordenadas: Listamos os dados em uma lista de tamanho n.

Listas Ordenadas: Ordenamos em uma lista de tamanho n os dados de acordo com suas prioridades em ordem decrescente.

	Lista Não Ordenadas	Lista Ordenada
Seleção	$O(n)$	$O(1)$
Inserção	$O(1)$	$O(n)$
Remoção	$O(n)$	$O(1)$
Alteração	$O(n)$	$O(n)$
Construção	$O(n)$	

Veremos um método mais eficiente chamado **heap**.

Listas Ordenadas e Não ordenadas

Listas Não Ordenadas: Listamos os dados em uma lista de tamanho n.

Listas Ordenadas: Ordenamos em uma lista de tamanho n os dados de acordo com suas prioridades em ordem decrescente.

	Lista Não Ordenadas	Lista Ordenada
Seleção	$O(n)$	$O(1)$
Inserção	$O(1)$	$O(n)$
Remoção	$O(n)$	$O(1)$
Alteração	$O(n)$	$O(n)$
Construção	$O(n)$	$O(n \log n)$

Veremos um método mais eficiente chamado **heap**.

Sumário

Heaps

- **Definição:** É uma lista linear H de tamanho n que satisfaz a seguinte propriedade:

$$\forall\ i > 1, H[i].chave\ \leq\ H[\lfloor i/2 \rfloor].chave.$$

Heaps

- **Definição:** É uma lista linear H de tamanho n que satisfaz a seguinte propriedade:

$$\forall\ i > 1, H[i].chave\ \leq\ H[\lfloor i/2 \rfloor].chave.$$

- O campo $H[i].chave$ representa a prioridade do dado i.

Heaps

- **Definição:** É uma lista linear H de tamanho n que satisfaz a seguinte propriedade:

$$\forall\ i > 1, H[i].chave\ \leq\ H[\lfloor i/2 \rfloor].chave.$$

- O campo $H[i].chave$ representa a prioridade do dado i.

ou de forma equivalente:

Heaps

- **Definição:** É uma lista linear H de tamanho n que satisfaz a seguinte propriedade:

$$\forall\ i > 1, H[i].chave\ \leq\ H[\lfloor i/2 \rfloor].chave.$$

- O campo $H[i].chave$ representa a prioridade do dado i.

ou de forma equivalente:

  - $H[i].chave\ \geq\ H[2i].chave$, para todo i tal que $2i \leq n$ **e**
  - $H[i].chave\ \geq\ H[2i+1].chave$, para todo i tal que $2i+1 \leq n$.

Heaps

- **Definição:** É uma lista linear H de tamanho n que satisfaz a seguinte propriedade:

$$\forall\ i > 1, H[i].chave\ \leq\ H[\lfloor i/2 \rfloor].chave.$$

- O campo $H[i].chave$ representa a prioridade do dado i.

ou de forma equivalente:

- $H[i].chave\ \geq\ H[2i].chave$, para todo i tal que $2i \leq n$ **e**
- $H[i].chave\ \geq\ H[2i+1].chave$, para todo i tal que $2i+1 \leq n$.

64	50	43	35	7	12	18	25	20
1	2	3	4	5	6	7	8	9

Heaps

- **Definição:** É uma lista linear H de tamanho n que satisfaz a seguinte propriedade:

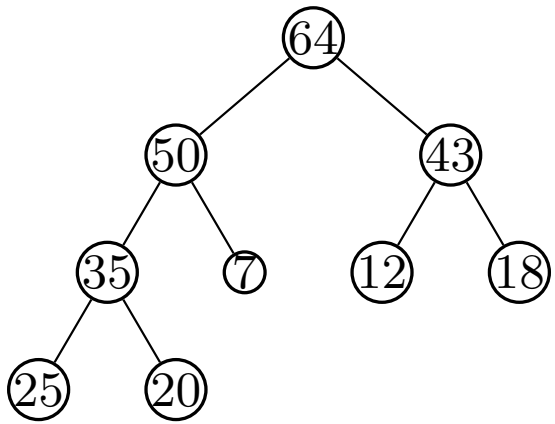
$$\forall\ i > 1, H[i].chave\ \leq\ H[\lfloor i/2 \rfloor].chave.$$

- O campo $H[i].chave$ representa a prioridade do dado i.

ou de forma equivalente:

- $H[i].chave\ \geq\ H[2i].chave$, para todo i tal que $2i \leq n$ **e**
- $H[i].chave\ \geq\ H[2i+1].chave$, para todo i tal que $2i+1 \leq n$.

64	50	43	35	7	12	18	25	20
1	2	3	4	5	6	7	8	9



Heaps

- **Definição:** É uma lista linear H de tamanho n que satisfaz a seguinte propriedade:

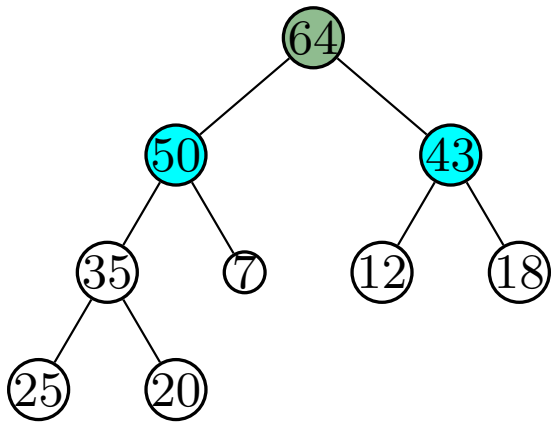
$$\forall\ i > 1, H[i].chave\ \leq\ H[\lfloor i/2 \rfloor].chave.$$

- O campo $H[i].chave$ representa a prioridade do dado i.

ou de forma equivalente:

- $H[i].chave\ \geq\ H[2i].chave$, para todo i tal que $2i \leq n$ **e**
- $H[i].chave\ \geq\ H[2i+1].chave$, para todo i tal que $2i+1 \leq n$.

64	50	43	35	7	12	18	25	20
1	2	3	4	5	6	7	8	9



Heaps

- **Definição:** É uma lista linear H de tamanho n que satisfaz a seguinte propriedade:

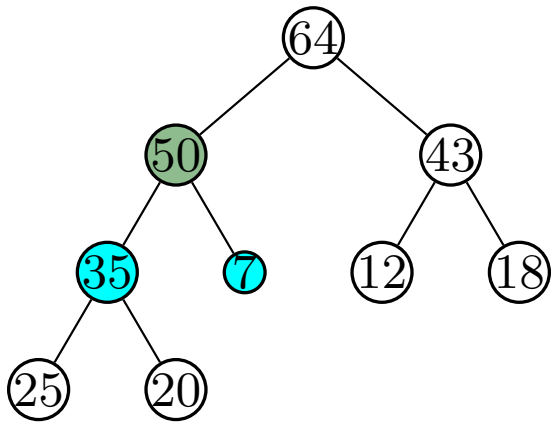
$$\forall\ i > 1, H[i].chave\ \leq\ H[\lfloor i/2 \rfloor].chave.$$

- O campo $H[i].chave$ representa a prioridade do dado i.

ou de forma equivalente:

- $H[i].chave\ \geq\ H[2i].chave$, para todo i tal que $2i \leq n$ **e**
- $H[i].chave\ \geq\ H[2i+1].chave$, para todo i tal que $2i+1 \leq n$.

64	50	43	35	7	12	18	25	20
1	2	3	4	5	6	7	8	9



Heaps

- **Definição:** É uma lista linear H de tamanho n que satisfaz a seguinte propriedade:

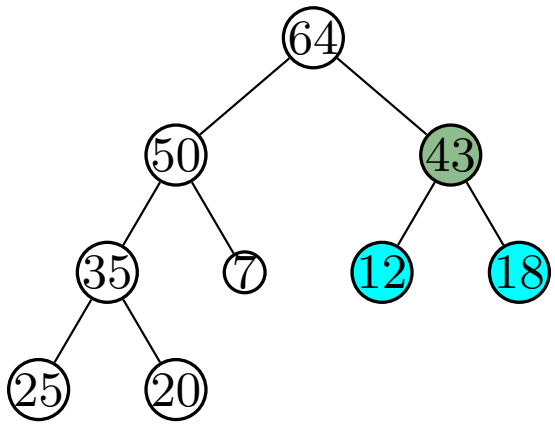
$$\forall\ i > 1, H[i].chave\ \leq\ H[\lfloor i/2 \rfloor].chave.$$

- O campo $H[i].chave$ representa a prioridade do dado i.

ou de forma equivalente:

- $H[i].chave\ \geq\ H[2i].chave$, para todo i tal que $2i \leq n$ **e**
- $H[i].chave\ \geq\ H[2i+1].chave$, para todo i tal que $2i+1 \leq n$.

64	50	43	35	7	12	18	25	20
1	2	3	4	5	6	7	8	9



Heaps

- **Definição:** É uma lista linear H de tamanho n que satisfaz a seguinte propriedade:

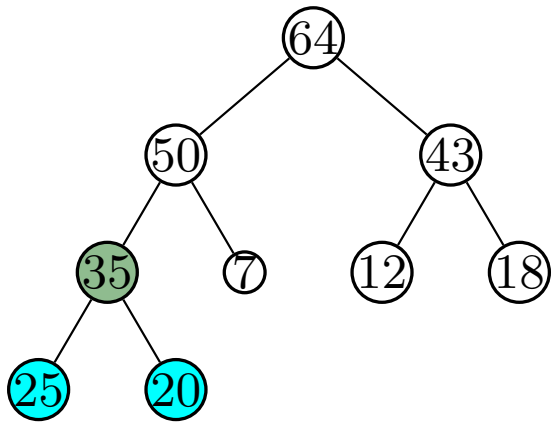
$$\forall\ i > 1, H[i].chave\ \leq\ H[\lfloor i/2 \rfloor].chave.$$

- O campo $H[i].chave$ representa a prioridade do dado i.

ou de forma equivalente:

- $H[i].chave\ \geq\ H[2i].chave$, para todo i tal que $2i \leq n$ **e**
- $H[i].chave\ \geq\ H[2i+1].chave$, para todo i tal que $2i+1 \leq n$.

64	50	43	35	7	12	18	25	20
1	2	3	4	5	6	7	8	9



Propriedades Sobre Heaps

- Estudaremos agora a complexidade de cada operação sobre heaps.

Propriedades Sobre Heaps

- Estudaremos agora a complexidade de cada operação sobre heaps.

	Lista Não Ordenadas	Lista Ordenada	Heap
Seleção	$O(n)$	$O(1)$	$O(1)$
Inserção	$O(1)$	$O(n)$	$O(\log n)$
Remoção	$O(n)$	$O(1)$	$O(\log n)$
Alteração	$O(n)$	$O(n)$	$O(\log n)$
Construção	$O(n)$	$O(n \log n)$	$O(n)$

Propriedades Sobre Heaps

- Estudaremos agora a complexidade de cada operação sobre heaps.

	Lista Não Ordenadas	Lista Ordenada	Heap
Seleção	$O(n)$	$O(1)$	$O(1)$
Inserção	$O(1)$	$O(n)$	$O(\log n)$
Remoção	$O(n)$	$O(1)$	$O(\log n)$
Alteração	$O(n)$	$O(n)$	$O(\log n)$
Construção	$O(n)$	$O(n \log n)$	$O(n)$

- A condição $\forall\ i > 1, H[i] \leq H[\lfloor i/2 \rfloor]$ implica que o elemento de maior prioridade seja sempre o primeiro no heap (raiz na árvore).

Propriedades Sobre Heaps

- Estudaremos agora a complexidade de cada operação sobre heaps.

	Lista Não Ordenadas	Lista Ordenada	Heap
Seleção	$O(n)$	$O(1)$	$O(1)$
Inserção	$O(1)$	$O(n)$	$O(\log n)$
Remoção	$O(n)$	$O(1)$	$O(\log n)$
Alteração	$O(n)$	$O(n)$	$O(\log n)$
Construção	$O(n)$	$O(n \log n)$	$O(n)$

- A condição $\forall\ i > 1, H[i] \leq H[\lfloor i/2 \rfloor]$ implica que o elemento de maior prioridade seja sempre o primeiro no heap (raiz na árvore).

- Logo a complexidade de seleção (busca) em heaps é sempre $O(1)$.

Alteração de Prioridade

- Podemos **aumentar** ou **diminuir** a prioridade de um elemento.

Alteração de Prioridade

- Podemos **aumentar** ou **diminuir** a prioridade de um elemento.
- Ao modificar uma prioridade de um elemento, queremos manter a propriedade de heap para a nova lista.

Alteração de Prioridade

- Podemos **aumentar** ou **diminuir** a prioridade de um elemento.
- Ao modificar uma prioridade de um elemento, queremos manter a propriedade de heap para a nova lista.
- O aumento está associado à "subida" do nó na árvore.

Alteração de Prioridade

- Podemos **aumentar** ou **diminuir** a prioridade de um elemento.
- Ao modificar uma prioridade de um elemento, queremos manter a propriedade de heap para a nova lista.
- O aumento está associado à "subida" do nó na árvore.
- A diminuição está associado à "descida" do nó na árvore.

Alteração de Prioridade

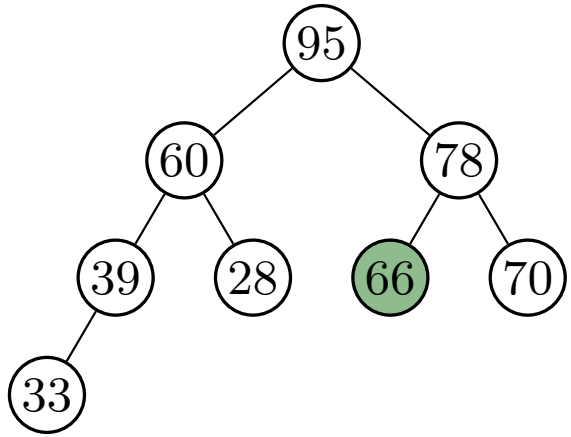
- Podemos **aumentar** ou **diminuir** a prioridade de um elemento.
- Ao modificar uma prioridade de um elemento, queremos manter a propriedade de heap para a nova lista.
- O aumento está associado à "subida" do nó na árvore.
- A diminuição está associado à "descida" do nó na árvore.
- A subida e descida serão realizados sempre através de caminhos.

Aumento de Prioridade

- Vamos aumentar a prioridade no nó 6 de 66 para 98.

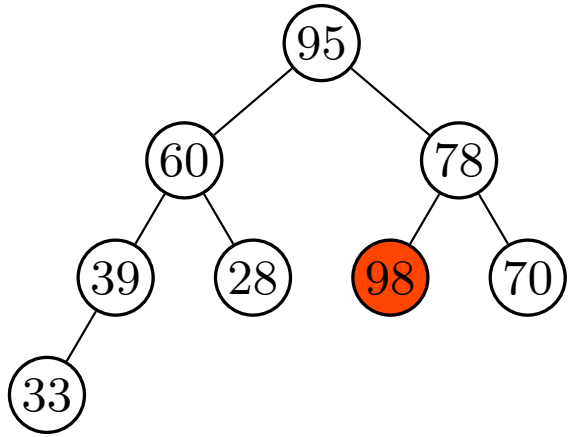
Aumento de Prioridade

- Vamos aumentar a prioridade no nó 6 de 66 para 98.



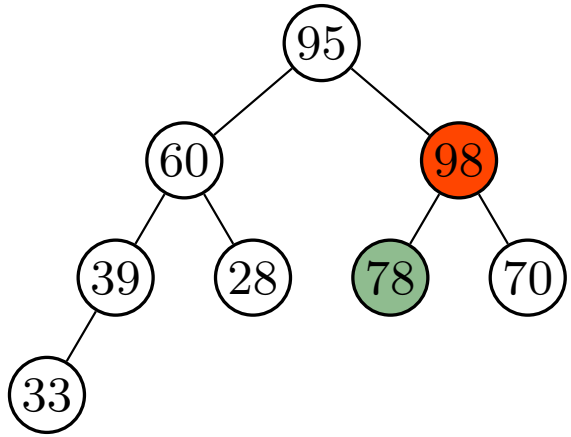
Aumento de Prioridade

- Vamos aumentar a prioridade no nó 6 de 66 para 98.



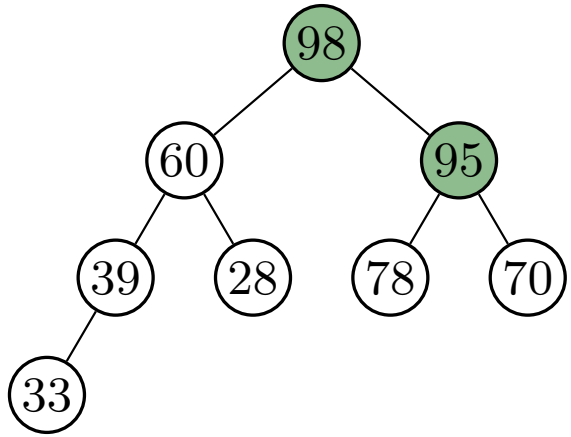
Aumento de Prioridade

- Vamos aumentar a prioridade no nó 6 de 66 para 98.



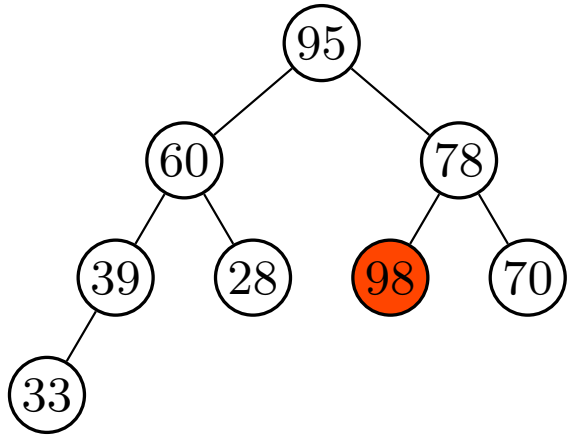
Aumento de Prioridade

- Vamos aumentar a prioridade no nó 6 de 66 para 98.



Aumento de Prioridade

- Vamos aumentar a prioridade no nó 6 de 66 para 98.



Algoritmo 1: Subir(i)

Entrada: Posição i a ser alterada.

```
1 j ← ⌊i/2⌋        % j representa o índice do pai de nó i.;
2 se j ≥ 1 então
3 |   se H[i].chave > H[j].chave então
4 |   |   H[i] ⇔ H[j];
5 |   |   Subir(j);
```

Complexidade do Procedimento Subir

- A complexidade é dada pelo número de trocas entre um nó com seu pai na árvore.

Complexidade do Procedimento Subir

- A complexidade é dada pelo número de trocas entre um nó com seu pai na árvore.

Exercício 1: Uma árvore binária completa é aquela em que, para todo nó v, se v possui alguma subárvore vazia, então v está ou no último (maior) ou penúltimo nível. Mostre que a altura h de qualquer árvore completa com $n > 0$ nós é $h \leq 1 + \lfloor \log n \rfloor$.

Complexidade do Procedimento Subir

- A complexidade é dada pelo número de trocas entre um nó com seu pai na árvore.

Exercício 1: Uma árvore binária completa é aquela em que, para todo nó v, se v possui alguma subárvore vazia, então v está ou no último (maior) ou penúltimo nível. Mostre que a altura h de qualquer árvore completa com $n > 0$ nós é $h \leq 1 + \lfloor \log n \rfloor$.

Dica: Usar indução matemática no número de nós.

Complexidade do Procedimento Subir

- A complexidade é dada pelo número de trocas entre um nó com seu pai na árvore.

  Exercício 1: Uma árvore binária completa é aquela em que, para todo nó v, se v possui alguma subárvore vazia, então v está ou no último (maior) ou penúltimo nível. Mostre que a altura h de qualquer árvore completa com $n > 0$ nós é $h \leq 1 + \lfloor \log n \rfloor$.

  Dica: Usar indução matemática no número de nós.

- Portanto, o número de trocas é no máximo a altura da árvore que representa o heap.

Complexidade do Procedimento Subir

- A complexidade é dada pelo número de trocas entre um nó com seu pai na árvore.

  Exercício 1: Uma árvore binária completa é aquela em que, para todo nó v, se v possui alguma subárvore vazia, então v está ou no último (maior) ou penúltimo nível. Mostre que a altura h de qualquer árvore completa com $n > 0$ nós é $h \leq 1 + \lfloor \log n \rfloor$.

  Dica: Usar indução matemática no número de nós.

- Portanto, o número de trocas é no máximo a altura da árvore que representa o heap.

- Logo a complexidade de Subir é $O(\log n)$.

Diminuição de Prioridade

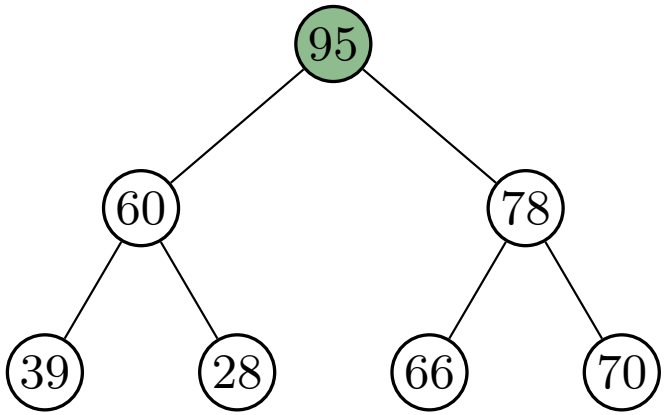
- A diminuição pode levar um nó i da árvore a possuir um filho com prioridade maior que a de i.

Diminuição de Prioridade

- A diminuição pode levar um nó i da árvore a possuir um filho com prioridade maior que a de i.
- Vamos diminuir a prioridade do nó 1 de 95 para 37.

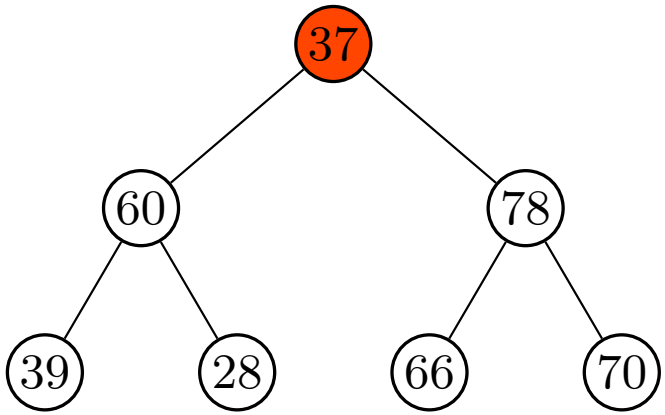
Diminuição de Prioridade

- A diminuição pode levar um nó i da árvore a possuir um filho com prioridade maior que a de i.
- Vamos diminuir a prioridade do nó 1 de 95 para 37.



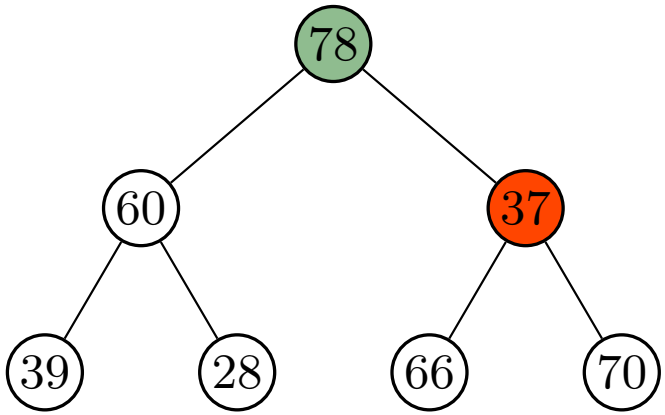
Diminuição de Prioridade

- A diminuição pode levar um nó i da árvore a possuir um filho com prioridade maior que a de i.
- Vamos diminuir a prioridade do nó 1 de 95 para 37.



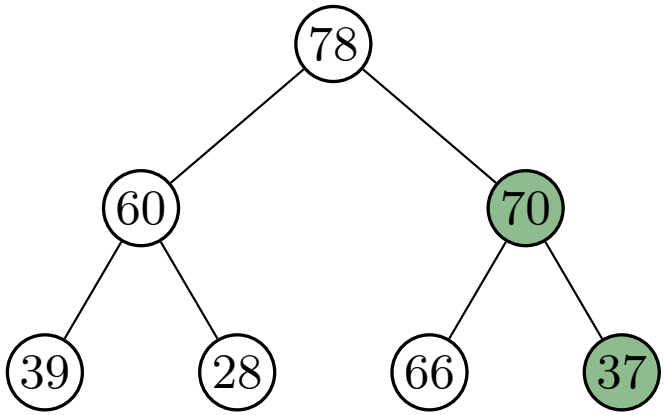
Diminuição de Prioridade

- A diminuição pode levar um nó i da árvore a possuir um filho com prioridade maior que a de i.
- Vamos diminuir a prioridade do nó 1 de 95 para 37.

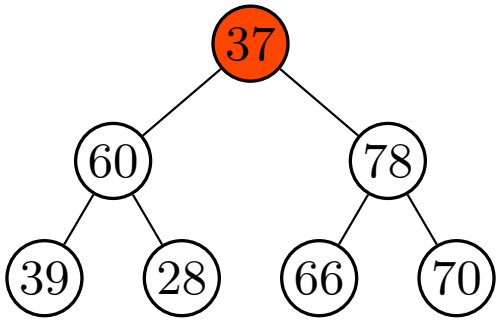


Diminuição de Prioridade

- A diminuição pode levar um nó i da árvore a possuir um filho com prioridade maior que a de i.
- Vamos diminuir a prioridade do nó 1 de 95 para 37.



Procedimento de Descida



Algoritmo 2: Descer(i, n)

Entrada: Posição i a ser alterada e número n de elementos do heap.

```
1  j ← 2i        % j representa o índice do filho esquerdo de i, se existir.;
2  se j ≤ n então
3      se j < n então
4          se H[j + 1].chave > H[j].chave então
5              j ← j + 1;
6      se H[i].chave < H[j].chave então
7          H[i] ⇔ H[j];
8          Descer(j, n);
```

Inserção e Remoção em Heaps

- **Inserção:**

Inserção e Remoção em Heaps

- **Inserção:**
  - adicionamos a nova prioridade ao heap de tamanho n na posição $n + 1$.

Inserção e Remoção em Heaps

- **Inserção:**
  - adicionamos a nova prioridade ao heap de tamanho n na posição $n + 1$.
  - A posição $n + 1$ é necessariamente uma folha na representação em árvore do heap.

Inserção e Remoção em Heaps

- **Inserção:**
  - adicionamos a nova prioridade ao heap de tamanho n na posição $n + 1$.
  - A posição $n + 1$ é necessariamente uma folha na representação em árvore do heap.
  - Aplicamos o procedimento Subir$(n + 1)$.

Inserção e Remoção em Heaps

- **Inserção:**
  - adicionamos a nova prioridade ao heap de tamanho n na posição $n+1$.
  - A posição $n+1$ é necessariamente uma folha na representação em árvore do heap.
  - Aplicamos o procedimento Subir$(n+1)$.
    - Complexidade: $O(\log n)$.

Inserção e Remoção em Heaps

- **Inserção:**
  - adicionamos a nova prioridade ao heap de tamanho n na posição $n+1$.
  - A posição $n+1$ é necessariamente uma folha na representação em árvore do heap.
  - Aplicamos o procedimento Subir$(n+1)$.
    - Complexidade: $O(\log n)$.
- **Remoção:**

Inserção e Remoção em Heaps

- **Inserção:**
  - adicionamos a nova prioridade ao heap de tamanho n na posição $n+1$.
  - A posição $n+1$ é necessariamente uma folha na representação em árvore do heap.
  - Aplicamos o procedimento Subir$(n+1)$.
    - Complexidade: $O(\log n)$.
- **Remoção:**
  - Lembre-se que a remoção e busca se dão **sempre** pelo elemento de **maior prioridade**, ou seja, o primeiro elemento do heap.

Inserção e Remoção em Heaps

- **Inserção:**
  - adicionamos a nova prioridade ao heap de tamanho n na posição $n + 1$.
  - A posição $n + 1$ é necessariamente uma folha na representação em árvore do heap.
  - Aplicamos o procedimento Subir$(n + 1)$.
    - Complexidade: $O(\log n)$.
- **Remoção:**
  - Lembre-se que a remoção e busca se dão **sempre** pelo elemento de **maior prioridade**, ou seja, o primeiro elemento do heap.
  - Remove-se o elemento da posição n para a primeira posição.

Inserção e Remoção em Heaps

- **Inserção:**
  - adicionamos a nova prioridade ao heap de tamanho n na posição $n + 1$.
  - A posição $n + 1$ é necessariamente uma folha na representação em árvore do heap.
  - Aplicamos o procedimento Subir$(n + 1)$.
    - Complexidade: $O(\log n)$.
- **Remoção:**
  - Lembre-se que a remoção e busca se dão **sempre** pelo elemento de **maior prioridade**, ou seja, o primeiro elemento do heap.
  - Remove-se o elemento da posição n para a primeira posição.
  - Decrementamos o número de elementos do heap para $n - 1$.

Inserção e Remoção em Heaps

- **Inserção:**
  - adicionamos a nova prioridade ao heap de tamanho n na posição $n+1$.
  - A posição $n+1$ é necessariamente uma folha na representação em árvore do heap.
  - Aplicamos o procedimento Subir$(n+1)$.
    - Complexidade: $O(\log n)$.
- **Remoção:**
  - Lembre-se que a remoção e busca se dão **sempre** pelo elemento de **maior prioridade**, ou seja, o primeiro elemento do heap.
  - Remove-se o elemento da posição n para a primeira posição.
  - Decrementamos o número de elementos do heap para $n-1$.
  - Aplica-se o procedimento Descer$(1, n)$.

Inserção e Remoção em Heaps

- **Inserção:**
  - adicionamos a nova prioridade ao heap de tamanho n na posição $n+1$.
  - A posição $n+1$ é necessariamente uma folha na representação em árvore do heap.
  - Aplicamos o procedimento Subir$(n+1)$.
    - Complexidade: $O(\log n)$.
- **Remoção:**
  - Lembre-se que a remoção e busca se dão **sempre** pelo elemento de **maior prioridade**, ou seja, o primeiro elemento do heap.
  - Remove-se o elemento da posição n para a primeira posição.
  - Decrementamos o número de elementos do heap para $n-1$.
  - Aplica-se o procedimento Descer$(1, n)$.
    - Complexidade: $O(\log n)$.

Construção de um Heap

Objetivo: obter um heap cujos elementos são os mesmos de um dado vetor V de tamanho n.

Construção de um Heap

Objetivo: obter um heap cujos elementos são os mesmos de um dado vetor V de tamanho n.

Solução 1: Ordenar o heap em ordem não crescente de prioridades.

Construção de um Heap

Objetivo: obter um heap cujos elementos são os mesmos de um dado vetor V de tamanho n.

Solução 1: Ordenar o heap em ordem não crescente de prioridades.

- Complexidade: $O(n \log n)$.

Construção de um Heap

Objetivo: obter um heap cujos elementos são os mesmos de um dado vetor V de tamanho n.

Solução 1: Ordenar o heap em ordem não crescente de prioridades.

- Complexidade: $O(n \log n)$.

Solução 2: Considera-se a construção do heap com as primeiras i prioridades de V e, a cada passo, insere-se o elemento $V[i]$ no heap atual.

Construção de um Heap

Objetivo: obter um heap cujos elementos são os mesmos de um dado vetor V de tamanho n.

Solução 1: Ordenar o heap em ordem não crescente de prioridades.

- Complexidade: $O(n \log n)$.

Solução 2: Considera-se a construção do heap com as primeiras i prioridades de V e, a cada passo, insere-se o elemento $V[i]$ no heap atual.

- Complexidade: $O(n \log n)$.

Algoritmo Linear para Construção de um Heap

- A propriedade de heap é sempre satisfeita para as **folhas** de qualquer vetor de entrada V.

Algoritmo Linear para Construção de um Heap

- A propriedade de heap é sempre satisfeita para as **folhas** de qualquer vetor de entrada V.
- Um nó folha não possui filhos, logo a propriedade de heaps é trivialmente satisfeita para elas.

Algoritmo Linear para Construção de um Heap

- A propriedade de heap é sempre satisfeita para as **folhas** de qualquer vetor de entrada V.
- Um nó folha não possui filhos, logo a propriedade de heaps é trivialmente satisfeita para elas.
- Exercício 2: Mostre que toda árvore binária completa possui ao menos metade de seus nós como nós folhas.

Algoritmo Linear para Construção de um Heap

- A propriedade de heap é sempre satisfeita para as **folhas** de qualquer vetor de entrada V.
- Um nó folha não possui filhos, logo a propriedade de heaps é trivialmente satisfeita para elas.
- Exercício 2: Mostre que toda árvore binária completa possui ao menos metade de seus nós como nós folhas.
- Logo as posições de $\lfloor n/2 \rfloor + 1$ até n constituem heaps independentes.

Algoritmo Linear para Construção de um Heap

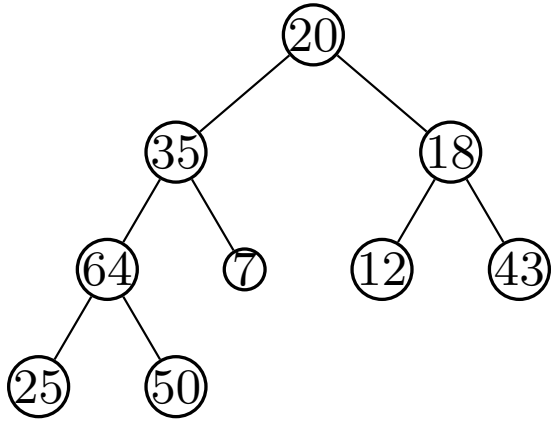
- A propriedade de heap é sempre satisfeita para as **folhas** de qualquer vetor de entrada V.
- Um nó folha não possui filhos, logo a propriedade de heaps é trivialmente satisfeita para elas.
- Exercício 2: Mostre que toda árvore binária completa possui ao menos metade de seus nós como nós folhas.
- Logo as posições de $\lfloor n/2 \rfloor + 1$ até n constituem heaps independentes.
- A ideia do algoritmo é considerar um vetor H de n posições cujas primeiras $\lfloor n/2 \rfloor$ posições são todas iguais ao maior elemento de V, enquanto as demais posições são idênticas às correspondentes em V.

Algoritmo Linear para Construção de um Heap

- A propriedade de heap é sempre satisfeita para as **folhas** de qualquer vetor de entrada V.
- Um nó folha não possui filhos, logo a propriedade de heaps é trivialmente satisfeita para elas.
- Exercício 2: Mostre que toda árvore binária completa possui ao menos metade de seus nós como nós folhas.
- Logo as posições de $\lfloor n/2 \rfloor + 1$ até n constituem heaps independentes.
- A ideia do algoritmo é considerar um vetor H de n posições cujas primeiras $\lfloor n/2 \rfloor$ posições são todas iguais ao maior elemento de V, enquanto as demais posições são idênticas às correspondentes em V.
- Para cada posição i de $\lfloor n/2 \rfloor$ até a primeira (nós internos), alteramos (diminuição) a prioridade de $H[i]$ para $V[i]$.

Exemplo

20	35	18	64	7	12	43	25	50
1	2	3	4	5	6	7	8	9



Algoritmo: ConstroiHeap(H, n)

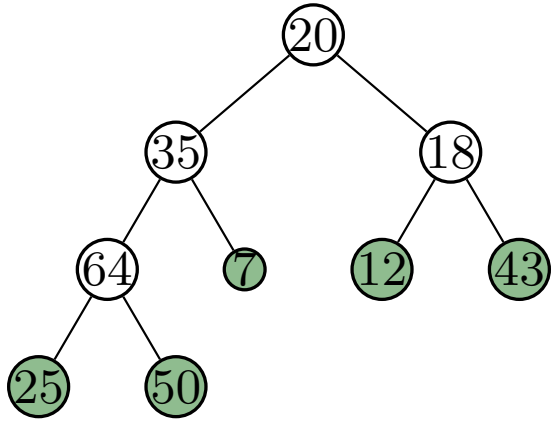
Entrada: Vetor de entrada $V = H$ de tamanho n.

1 **para** $i \leftarrow \lfloor n/2 \rfloor, \ldots, 1$ **faça**

2 Descer(i, n);

Exemplo

20	35	18	64	7	12	43	25	50
1	2	3	4	5	6	7	8	9



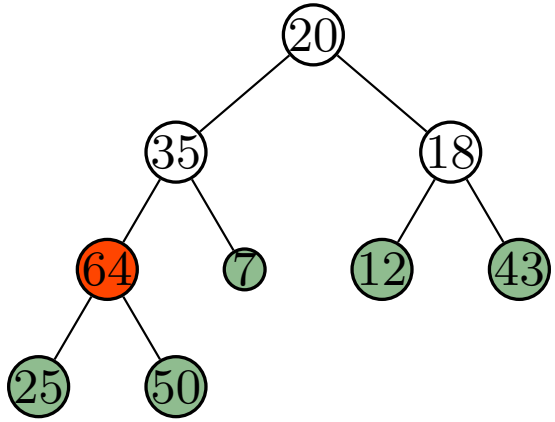
Algoritmo: ConstroiHeap(H, n)

Entrada: Vetor de entrada $V = H$ de tamanho n.

para $i \leftarrow \lfloor n/2 \rfloor, \ldots, 1$ **faça**
    Descer(i, n);

Exemplo

20	35	18	64	7	12	43	25	50
1	2	3	4	5	6	7	8	9



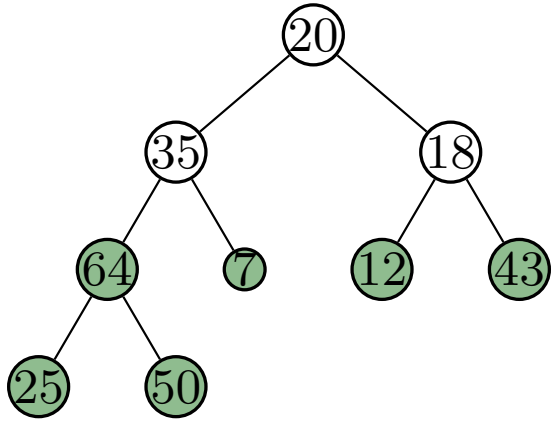
Algoritmo: ConstroiHeap(H, n)

Entrada: Vetor de entrada $V = H$ de tamanho n.

para $i \leftarrow \lfloor n/2 \rfloor, \ldots, 1$ **faça**
    Descer(i, n);

Exemplo

20	35	18	64	7	12	43	25	50
1	2	3	4	5	6	7	8	9



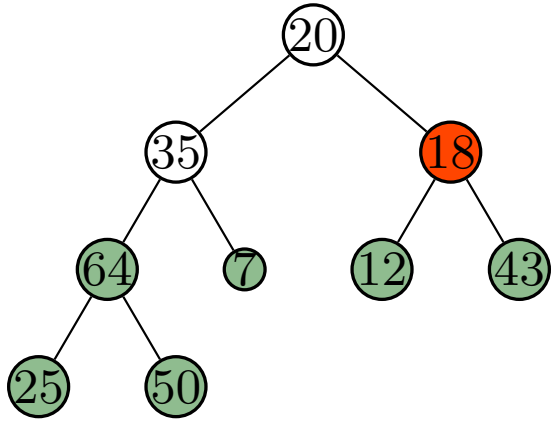
Algoritmo: ConstroiHeap(H, n)

Entrada: Vetor de entrada $V = H$ de tamanho n.

para $i \leftarrow \lfloor n/2 \rfloor, \ldots, 1$ **faça**
    Descer(i, n);

Exemplo

20	35	18	64	7	12	43	25	50
1	2	3	4	5	6	7	8	9



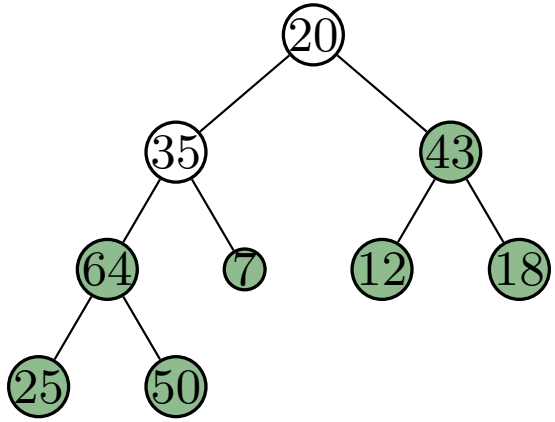
Algoritmo: ConstroiHeap(H, n)

Entrada: Vetor de entrada $V = H$ de tamanho n.

para $i \leftarrow \lfloor n/2 \rfloor, \ldots, 1$ **faça**
    Descer(i, n);

Exemplo

20	35	43	64	7	12	18	25	50
1	2	3	4	5	6	7	8	9



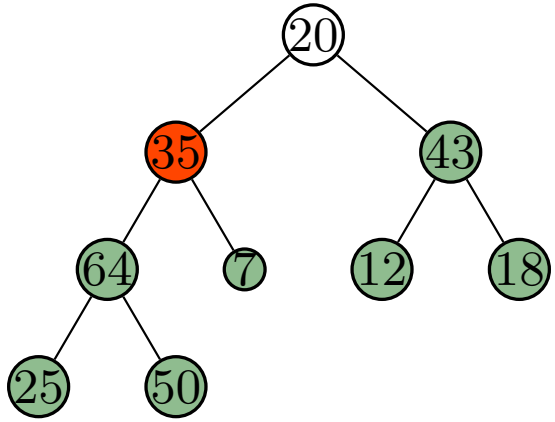
Algoritmo: ConstroiHeap(H, n)

Entrada: Vetor de entrada $V = H$ de tamanho n.

para $i \leftarrow \lfloor n/2 \rfloor, \ldots, 1$ **faça**
    Descer(i, n);

Exemplo

20	35	43	64	7	12	18	25	50
1	2	3	4	5	6	7	8	9



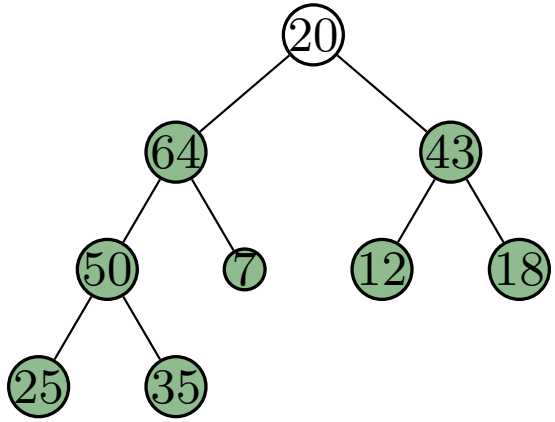
Algoritmo: ConstroiHeap(H, n)

Entrada: Vetor de entrada $V = H$ de tamanho n.

para $i \leftarrow \lfloor n/2 \rfloor, \ldots, 1$ **faça**
    Descer(i, n);

Exemplo

20	64	43	50	7	12	18	25	35
1	2	3	4	5	6	7	8	9



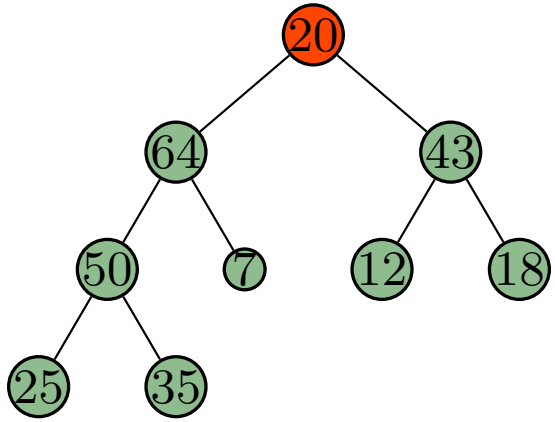
Algoritmo: ConstroiHeap(H, n)

Entrada: Vetor de entrada $V = H$ de tamanho n.

para $i \leftarrow \lfloor n/2 \rfloor, \ldots, 1$ **faça**
    Descer(i, n);

Exemplo

20	64	43	50	7	12	18	25	35
1	2	3	4	5	6	7	8	9



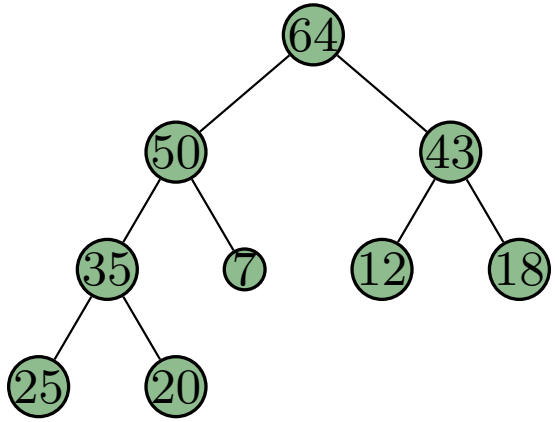
Algoritmo: ConstroiHeap(H, n)

Entrada: Vetor de entrada $V = H$ de tamanho n.

para $i \leftarrow \lfloor n/2 \rfloor, \ldots, 1$ **faça**
    Descer(i, n);

Exemplo

64	50	43	35	7	12	18	25	20
1	2	3	4	5	6	7	8	9



Algoritmo: ConstroiHeap(H, n)

Entrada: Vetor de entrada $V = H$ de tamanho n.

para $i \leftarrow \lfloor n/2 \rfloor, \ldots, 1$ **faça**
    Descer(i, n);

Exemplo

- **Exercício 3:** Mostre que a complexidade do Algoritmo ConstroiHeap é $O(n)$.

Aplicação de Heaps: Heapsort

- Seja V um vetor de tamanho n ao qual se quer ordenar em ordem não decrescente.

Aplicação de Heaps: Heapsort

- Seja V um vetor de tamanho n ao qual se quer ordenar em ordem não decrescente.
- Construímos um heap H a partir de V ($O(n)$).

Aplicação de Heaps: Heapsort

- Seja V um vetor de tamanho n ao qual se quer ordenar em ordem não decrescente.
- Construímos um heap H a partir de V ($O(n)$).
- Iterativamente removemos o elemento de maior prioridade.

Aplicação de Heaps: Heapsort

- Seja V um vetor de tamanho n ao qual se quer ordenar em ordem não decrescente.
- Construímos um heap H a partir de V ($O(n)$).
- Iterativamente removemos o elemento de maior prioridade.
- Como vimos, a remoção do elemento de maior prioridade é feita através do Procedimento Descer através da substituição do elemento de maior prioridade pelo último elemento do Heap.

Aplicação de Heaps: Heapsort

- Seja V um vetor de tamanho n ao qual se quer ordenar em ordem não decrescente.
- Construímos um heap H a partir de V ($O(n)$).
- Iterativamente removemos o elemento de maior prioridade.
- Como vimos, a remoção do elemento de maior prioridade é feita através do Procedimento Descer através da substituição do elemento de maior prioridade pelo último elemento do Heap.
- Além disso, a última posição do heap deixa de pertencer ao heap, uma vez que removemos um de seus elementos.

Aplicação de Heaps: Heapsort

- Seja V um vetor de tamanho n ao qual se quer ordenar em ordem não decrescente.
- Construímos um heap H a partir de V ($O(n)$).
- Iterativamente removemos o elemento de maior prioridade.
- Como vimos, a remoção do elemento de maior prioridade é feita através do Procedimento Descer através da substituição do elemento de maior prioridade pelo último elemento do Heap.
- Além disso, a última posição do heap deixa de pertencer ao heap, uma vez que removemos um de seus elementos.
- O algoritmo Heapsort faz uso desta posição liberada pelo heap para armazenar o elemento removido.

Heapsort

Algoritmo: Heapsort(H, n)

Entrada: Vetor de entrada H de tamanho n.

```
1 ConstroiHeap(H, n);
2 m ← n;
3 enquanto m > 1 faça
4     x ← Remover(H, m);
5     H[m] ← x;
6     m ← m − 1;
```

Heapsort

Algoritmo: Heapsort(H, n)

Entrada: Vetor de entrada H de tamanho n.

```
1 ConstroiHeap(H, n);
2 m ← n;
3 enquanto m > 1 faça
4     x ← Remover(H, m);
5     H[m] ← x;
6     m ← m − 1;
```

- Complexidade: $O(n \log n)$.

Heapsort

Algoritmo: Heapsort(H, n)

Entrada: Vetor de entrada H de tamanho n.

```
1 ConstroiHeap(H, n);
2 m ← n;
3 enquanto m > 1 faça
4     x ← Remover(H, m);
5     H[m] ← x;
6     m ← m − 1;
```

- Complexidade: $O(n \log n)$.
- Observação: Podemos definir um heap cujo objetivo seja descobrir a menor prioridade de um vetor.

Heapsort

Algoritmo: Heapsort(H, n)

Entrada: Vetor de entrada H de tamanho n.

```
1 ConstroiHeap(H, n);
2 m ← n;
3 enquanto m > 1 faça
4 |   x ← Remover(H, m);
5 |   H[m] ← x;
6 |   m ← m − 1;
```

- Complexidade: $O(n \log n)$.
- Observação: Podemos definir um heap cujo objetivo seja descobrir a menor prioridade de um vetor.
- Tais heaps são chamados de heaps de mínimo, enquanto os que vimos são conhecidos como heaps de máximo.