ANÁLISE DAS ESTRUTURAS DE PRIORIDADE APLICADAS AO CONTROLE DE TRÁFEGO ÁEREO

EQUIPE 6: CARLOS ESTELLITA, CRISLANE MARIA, DAVI DE PAULA, DOUGLAS ARAÚJO, WILGNER VIDAL.

DISCIPLINA MINISTRADA PELO PROFESSOR BRUNO SIMÕES

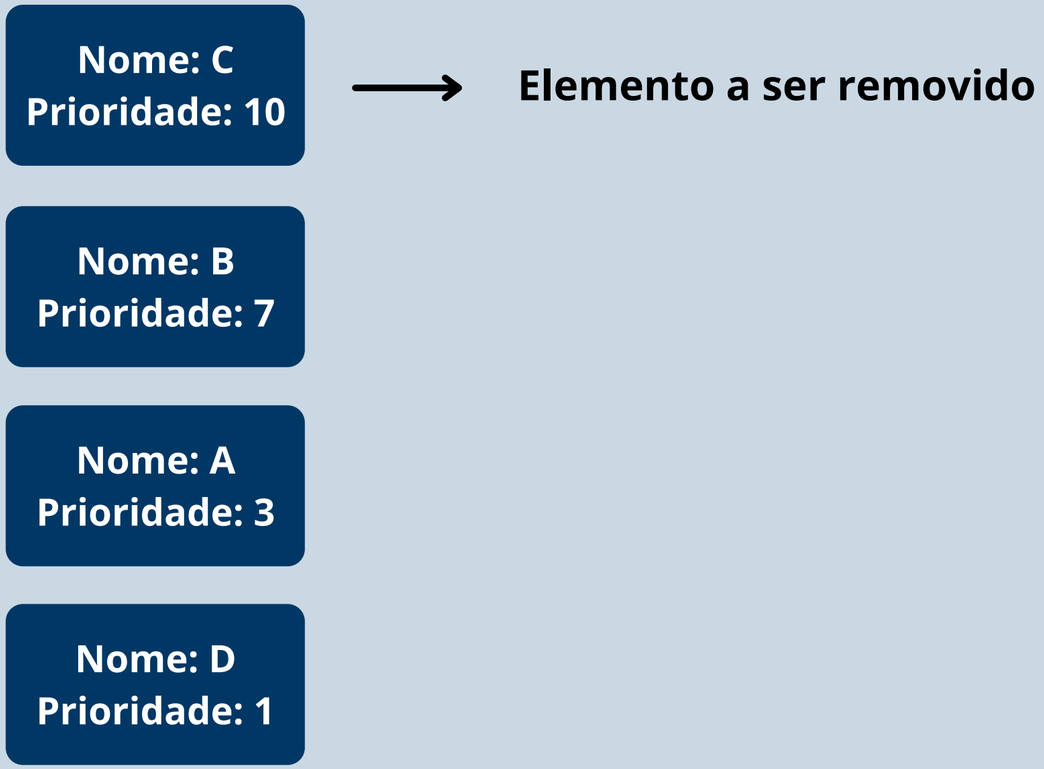
Sumário

Introdução

Neste presente estudo, foram aplicados os conceitos das estruturas de prioridade em uma simulação de tráfego aéreo com o intuito de identificar qual algorítmo possui maior eficácia nesta determinada situação.

Fundamentação Teórica

1. Fila de prioridade:



Fila de Prioridade

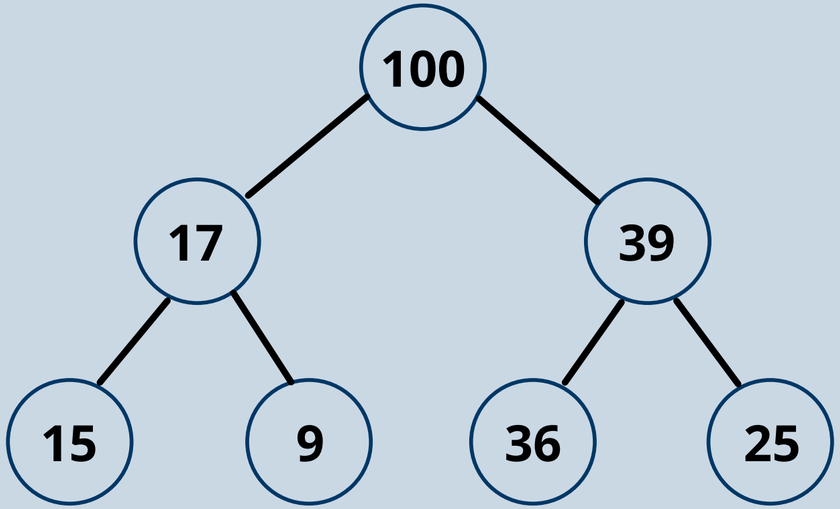
Fundamentação Teórica

2. Heap Binário:

- É um array que representa uma árvore binária completa ou quase completa.
- O filho esquerdo de um nó i é calculado por 2*i + 1, e o seu filho direito é calculado por 2*i + 2.
- No Heap Máximo os nós pais são sempre maiores que os seus filho, ao contrário, do Heap Mínimo onde os nós pais são sempre menores que os seus filhos.

Fundamentação Teórica

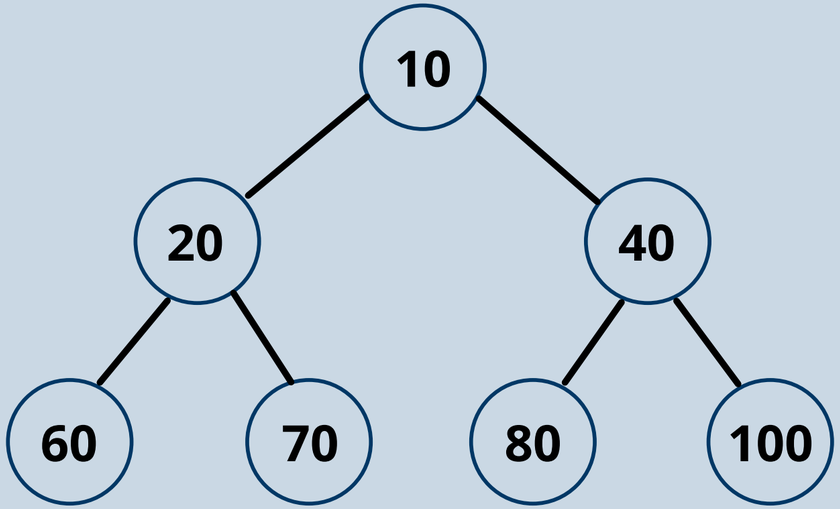
2. Heap Binário:



Heap Binário Máximo

Fundamentação Teórica

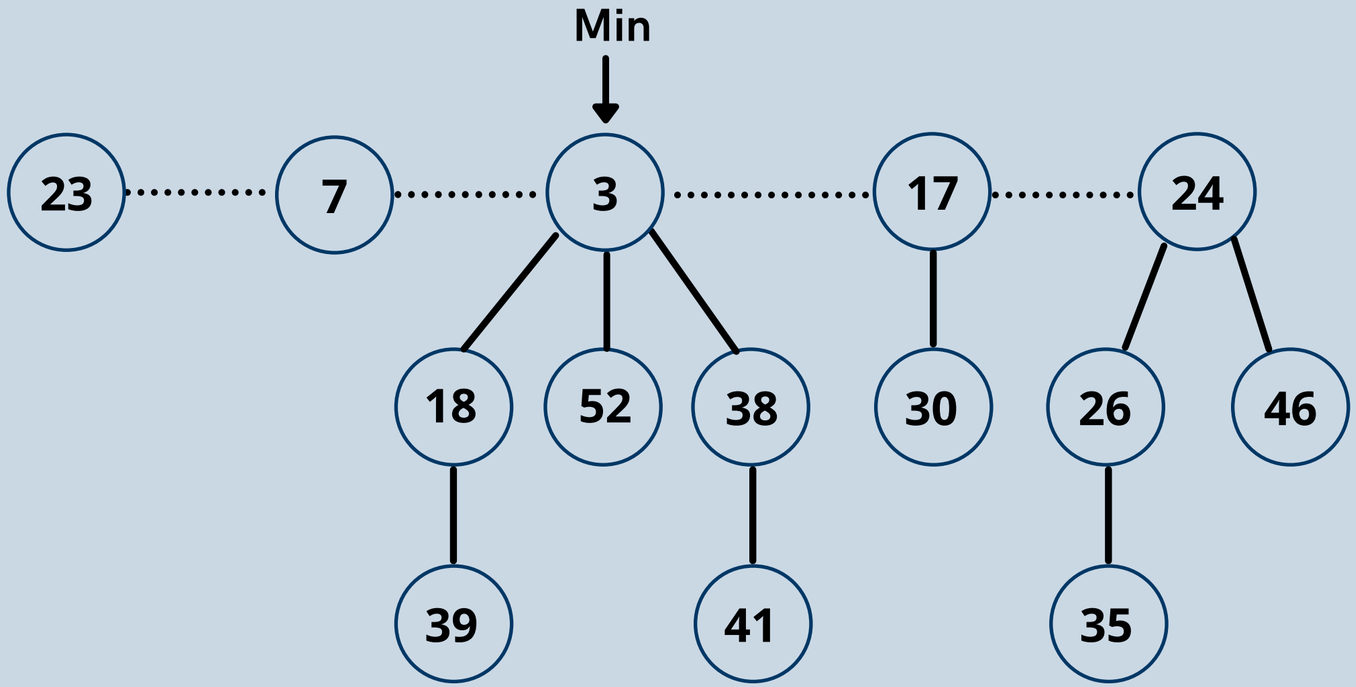
2. Heap Binário:



Heap Binário Minímo

Fundamentação Teórica

3. Heap Fibonacci:



Metodologia

1. Problema e Solução:

- O problema consiste em controlar o tráfego aéreo de aviões a partir dos algorítmos de prioridade com o intuito de verificar qual estrutura é mais pertinente para a situação. Para isso, implementamos a fila de prioridade usando: **array ordenado e desordenado, heap binária, heap de Fibonacci.**

- A solução obtida consiste em utilizar a fila de prioridade para estabelecer a precedência de pousos dos aviões de acordo o nível de combustível de cada um deles, ou seja, os aviões com menor nível de combustível terá a prioridade para pouso, com intuito de evitar acidentes por falta de abastecimento.

Metodologia

1. Problema e Solução:

Prioridade	Nome	Nível de Combustível
1	Airbus A380	11%
2	Boeing 737 MAX 7	27%
3	Embraer E-195	48%
4	Airbus A330	67%
5	Boeing 767	68%

Tabela 1. Quadro de prioridades de pousos

Metodologia

2. Classe Avião

```
class Aviao():
  def __init__(self, nome = None, prioridade = None) ->
None:
    self.nome = nome
    self.prioridade = prioridade

  def __repr__(self):
      rep = "Nome: " + str(self.nome) + " | " + "Nivel de
Combustivel: " + str(self.prioridade) + "%"
    return rep

  def __str__(self) -> str:
    return self.__repr__()
```

Metodologia

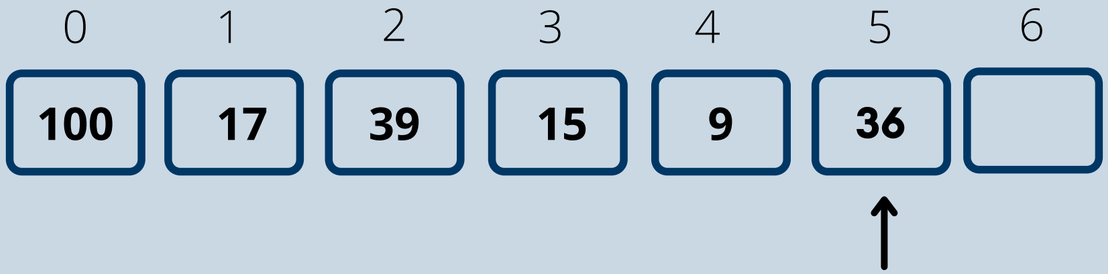
3. Vetor Desordenado - Inserir:

Inserindo o elemento de prioridade 36:

Metodologia

3. Vetor Desordenado - Inserir:

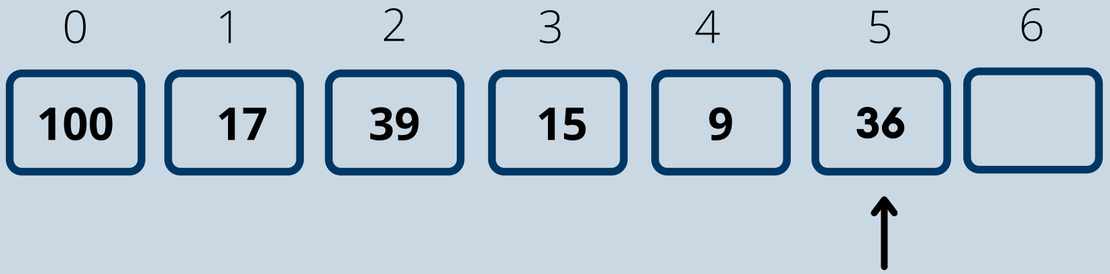
Inserindo o elemento de prioridade 36:



Metodologia

3. Vetor Desordenado - Inserir:

Inserindo o elemento de prioridade 36:



Complexidade: O(1)

Metodologia

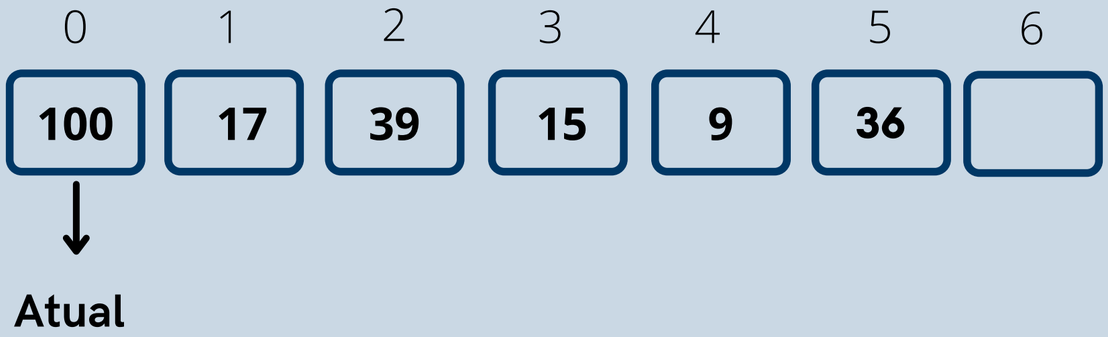
3. Vetor Desordenado - Inserir:

```
def inserir(self, nome, prioridade):
    # Verificando se a fila está cheia.
    if self.cheia():
      print("Fila Cheia.")
      return False

    # Criando o elemento (Aviao) a ser inserido.
    novo_dado = Aviao(nome, prioridade)

    # Inserindo o novo elemento na última posição.
    self.dados[self.quantidade] = novo_dado

    # Incrementando em uma unidade o valor quantidade.
    self.quantidade += 1

    return True
```

Metodologia

3. Vetor Desordenado - Remover:

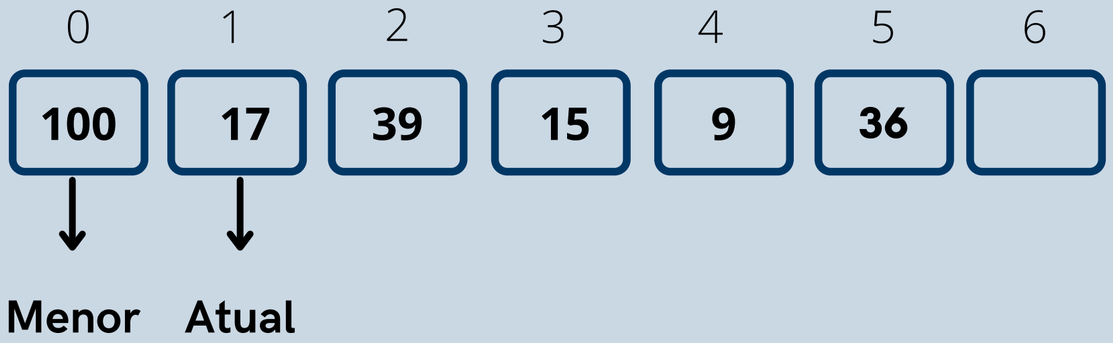
Remove o elemento de menor valor (maior prioridade):



Metodologia

3. Vetor Desordenado - Remover:

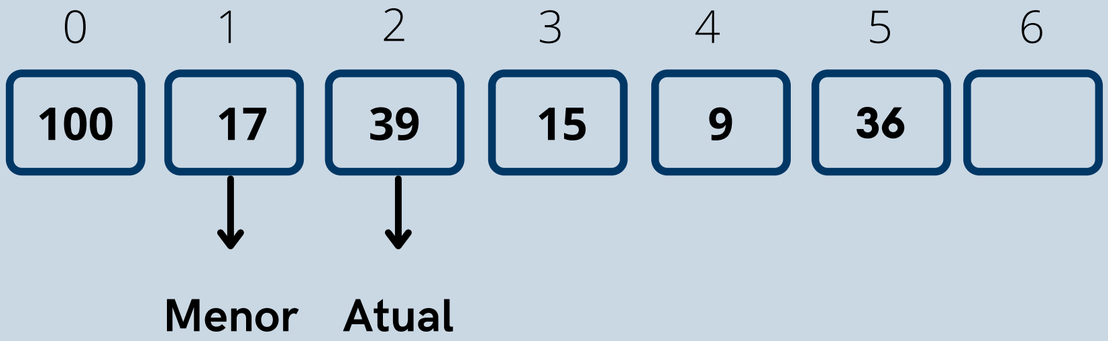
Remove o elemento de menor valor (maior prioridade):



Metodologia

3. Vetor Desordenado - Remover:

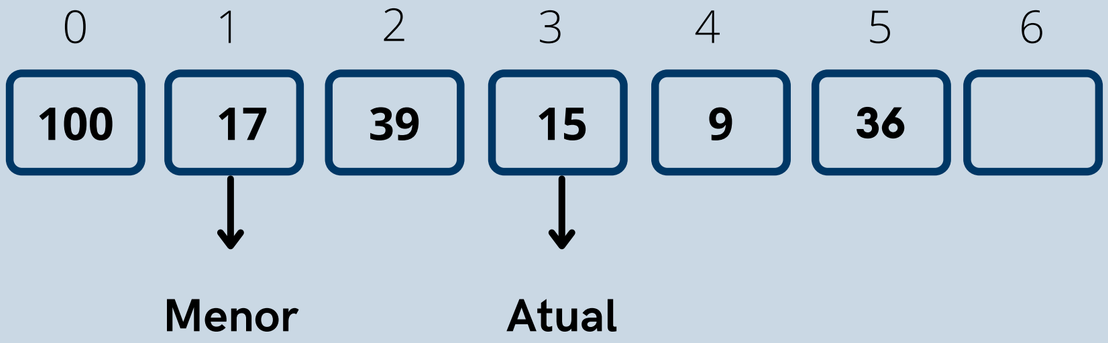
Remove o elemento de menor valor (maior prioridade):



Metodologia

3. Vetor Desordenado - Remover:

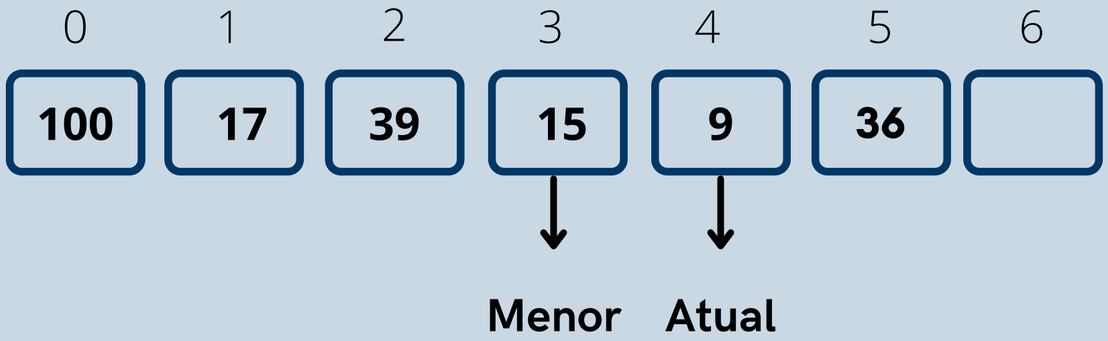
Remove o elemento de menor valor (maior prioridade):



Metodologia

3. Vetor Desordenado - Remover:

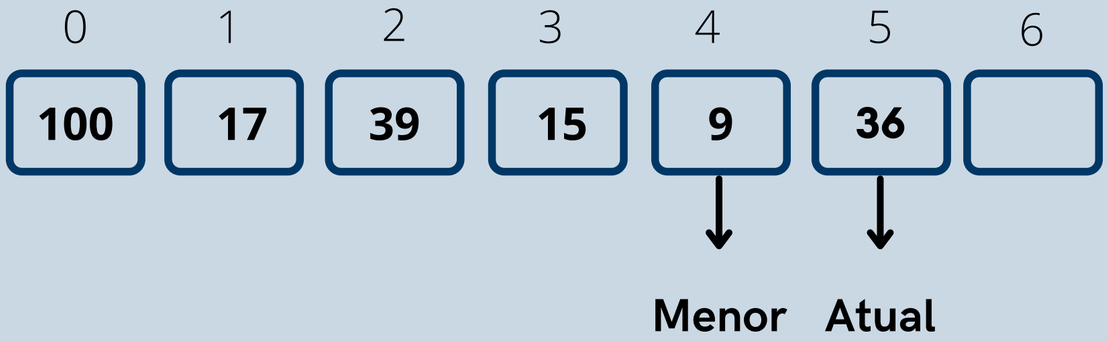
Remove o elemento de menor valor (maior prioridade):



Metodologia

3. Vetor Desordenado - Remover:

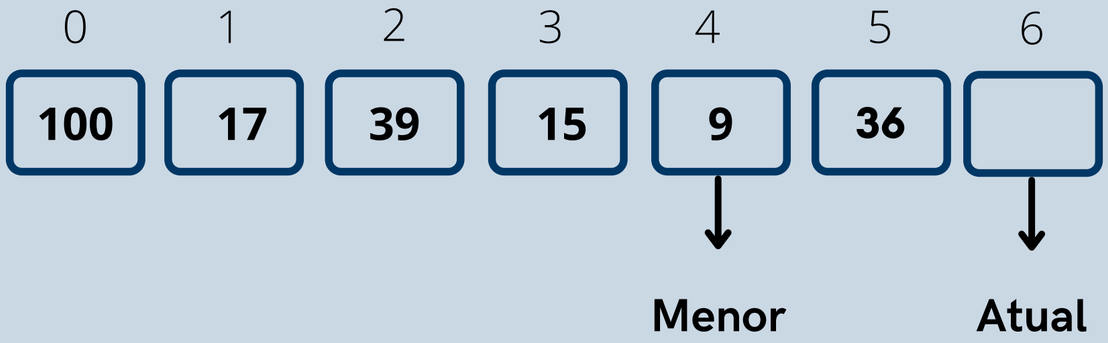
Remove o elemento de menor valor (maior prioridade):



Metodologia

3. Vetor Desordenado - Remover:

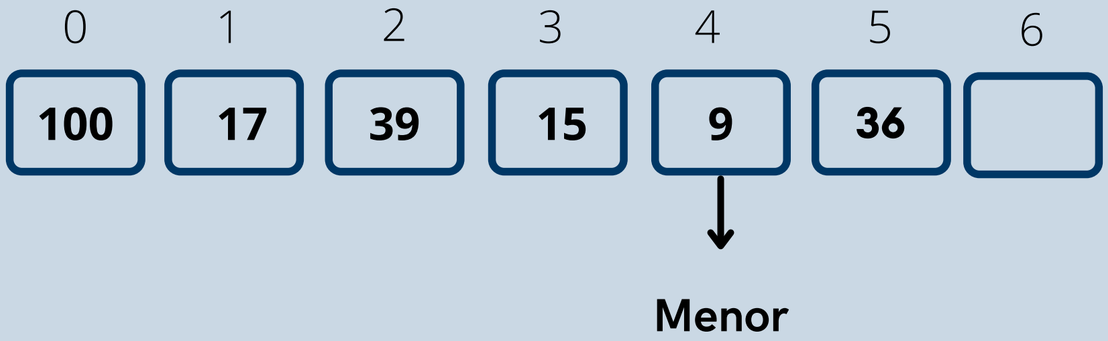
Remove o elemento de menor valor (maior prioridade):



Metodologia

3. Vetor Desordenado - Remover:

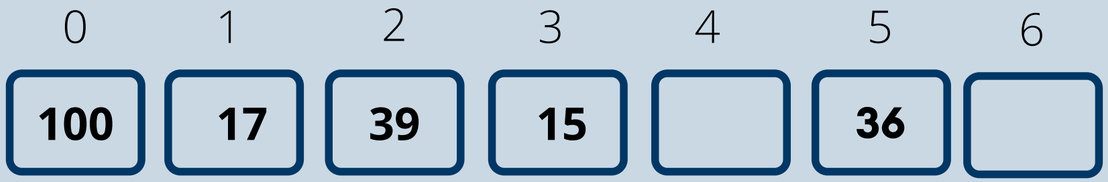
Remove o elemento de menor valor (maior prioridade):



Metodologia

3. Vetor Desordenado - Remover:

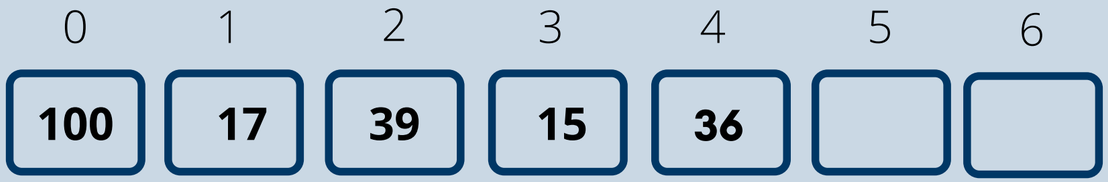
Remove o elemento de menor valor (maior prioridade):



Metodologia

3. Vetor Desordenado - Remover:

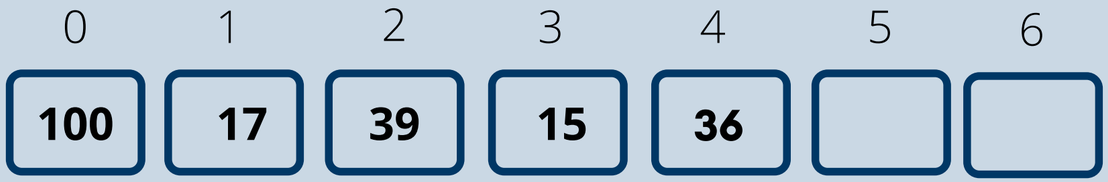
Remove o elemento de menor valor (maior prioridade):



Metodologia

3. Vetor Desordenado - Remover:

Remove o elemento de menor valor (maior prioridade):



Complexidade: O(n)

Metodologia

3. Vetor Desordenado - Remover

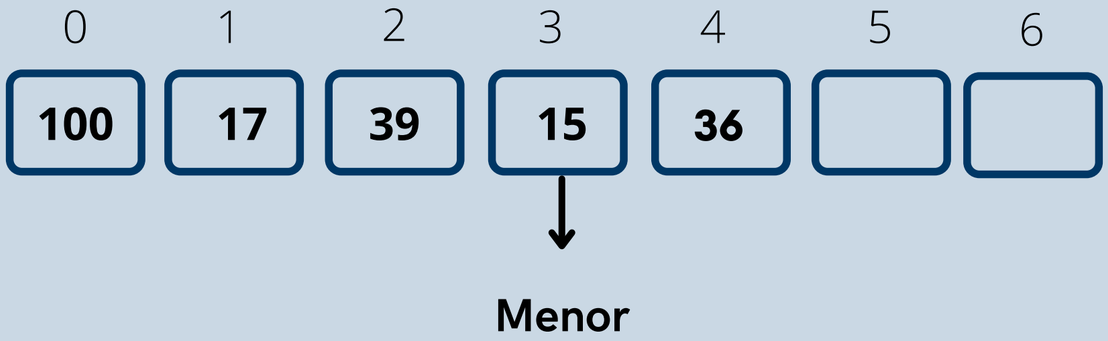
```
def remover(self):
# Verificando se a fila está vazia.
if self.vazia():
print("Fila Vazia.")
return False

# Procurando o elemento de menor prioridade.
menor = self.dados[0]
indice = 0

for i in range(1, self.quantidade):
if menor.prioridade > self.dados[i].prioridade:
menor = self.dados[i]
indice = i
```

```
# Preenchendo o elemento removido com seus sucessores.
for i in range(indice, self.quantidade - 1):
self.dados[i] = self.dados[i + 1]

# Decrementando em uma unidade o valor quantidade.
self.quantidade -= 1

return True
```

Metodologia

3. Vetor Desordenado - Consultar

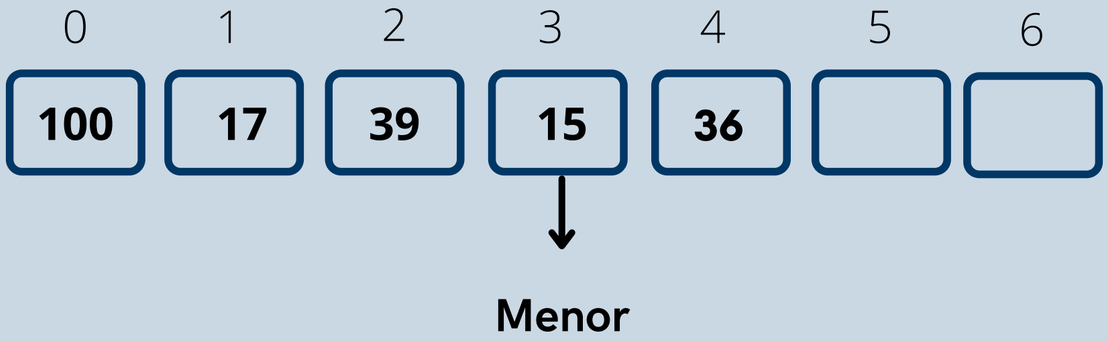
Consultando o elemento de menor valor (maior prioridade):



Metodologia

3. Vetor Desordenado - Consultar

Consultando o elemento de menor valor (maior prioridade):



Complexidade: O(n)

Metodologia

3. Vetor Desordenado - Consultar

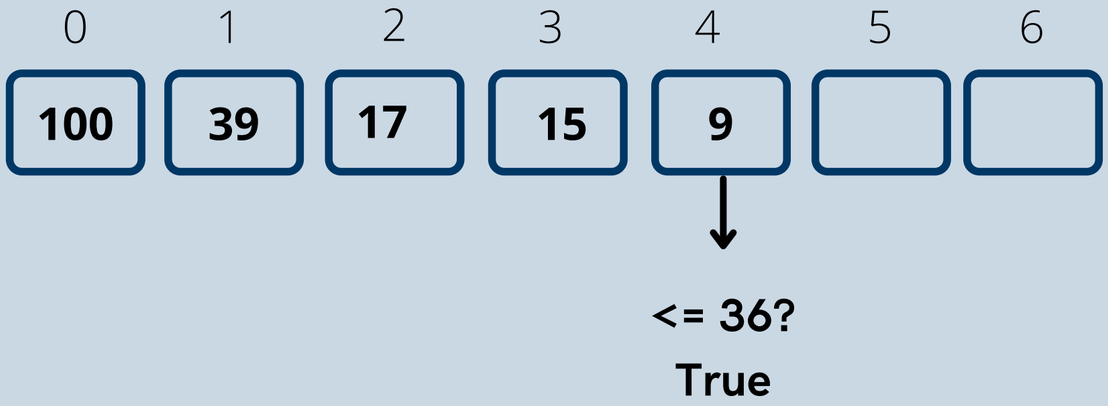
```
def consultar(self):
# Verificando se a fila está vazia.
if self.vazia():
print("Fila Vazia.")
return False

# Procurando o elemento de maior prioridade.
menor = self.dados[0]
indice = 0

for i in range(1, self.quantidade):
if menor.prioridade > self.dados[i].prioridade:
menor = self.dados[i]
indice = i
```

```
# Mostrando o elemento de maior prioridade
print(self.dados[indice])

return True
```

Metodologia

4. Vetor Ordenado - Inserir

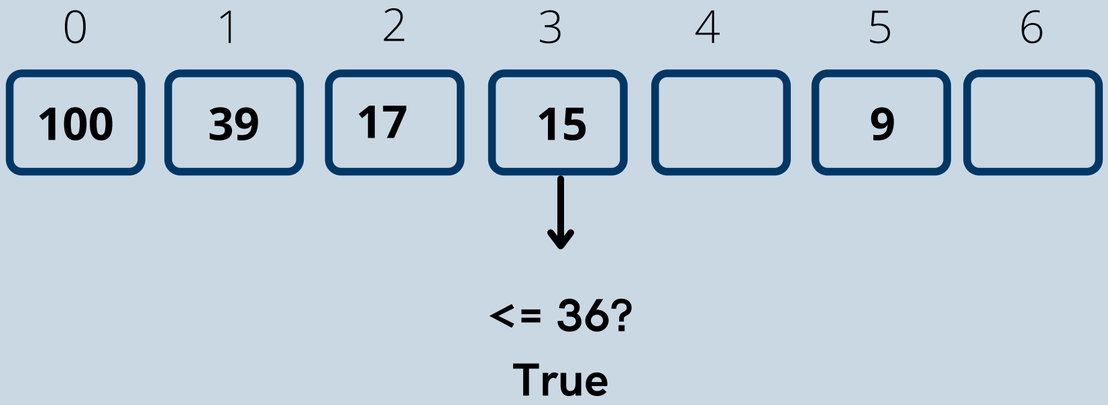
Inserindo o elemento de prioridade 36:



Metodologia

4. Vetor Ordenado - Inserir

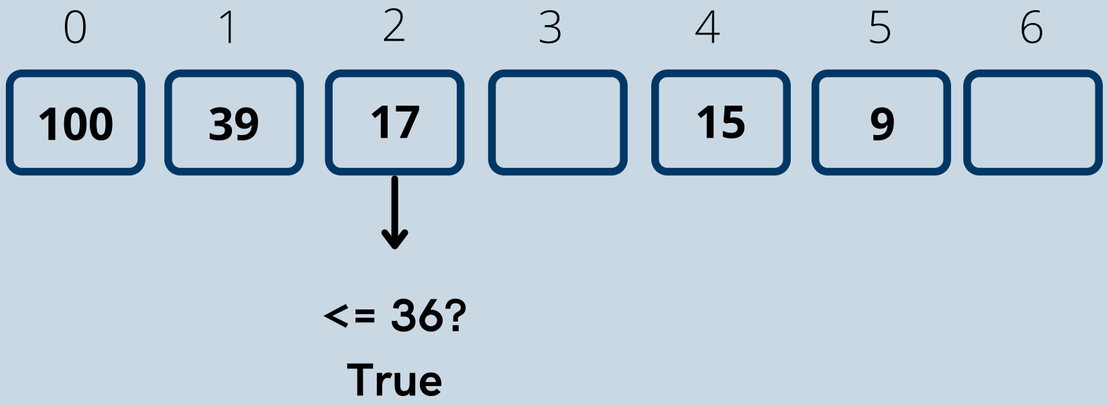
Inserindo o elemento de prioridade 36:



Metodologia

4. Vetor Ordenado - Inserir

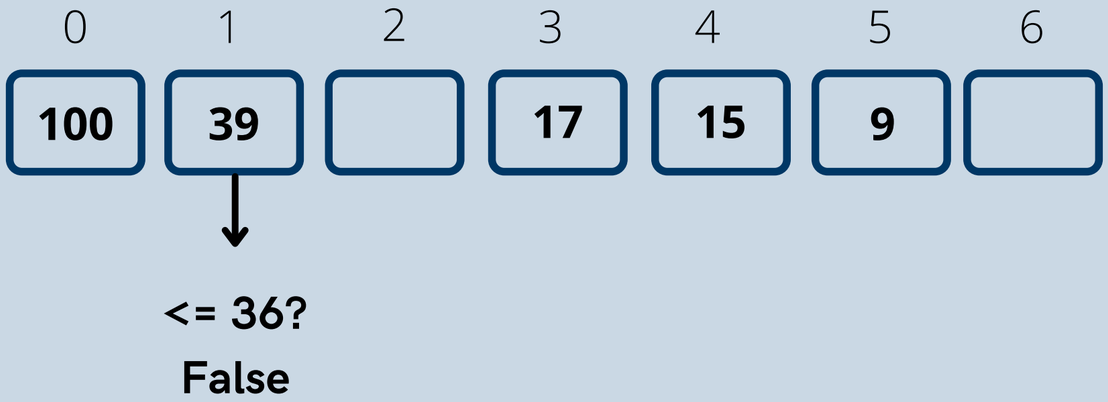
Inserindo o elemento de prioridade 36:



Metodologia

4. Vetor Ordenado - Inserir

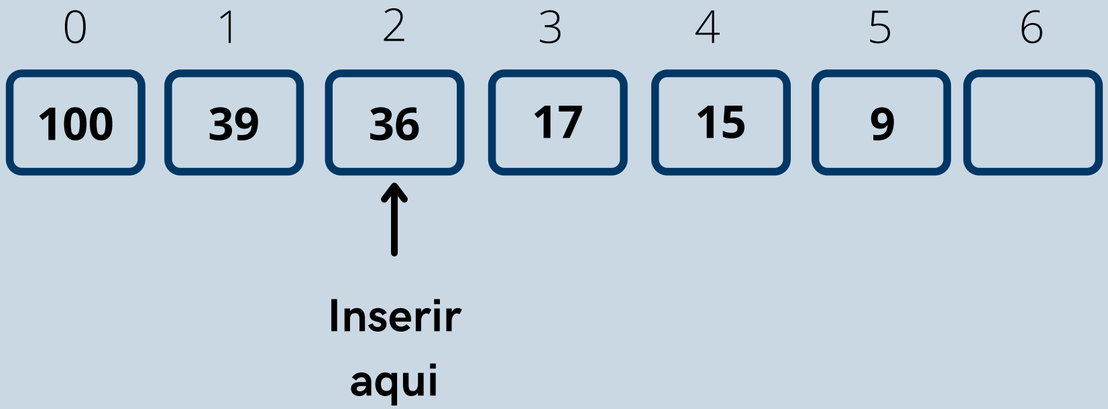
Inserindo o elemento de prioridade 36:



Metodologia

4. Vetor Ordenado - Inserir

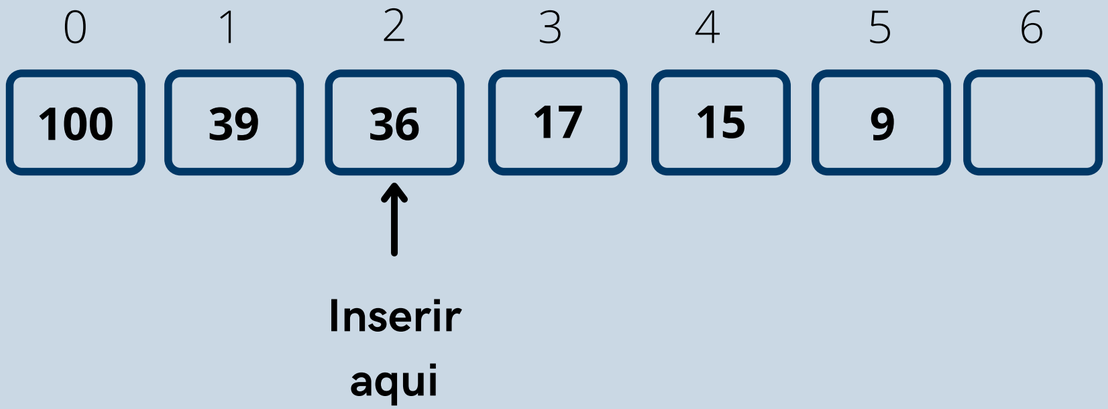
Inserindo o elemento de prioridade 36:



Metodologia

4. Vetor Ordenado - Inserir

Inserindo o elemento de prioridade 36:



Complexidade: O(n)

Metodologia

4. Vetor Ordenado - Inserir

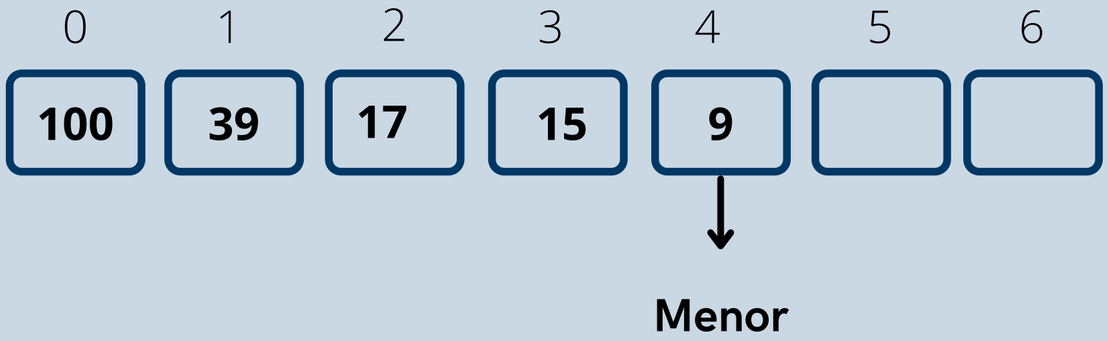
```
def inserir(self, nome, prioridade):
 # Verificando se a fila está cheia.
  if self.cheia():
    print("Fila Cheia.")
    return False

  # Criando variável para iterar no loop.
  i = self.quantidade - 1

  # Liberando a posição correta para inserir o novo elemento.
  while i >= 0 and self.dados[i].prioridade <= prioridade:
    self.dados[i + 1] = self.dados[i]
    i -= 1
```

```
# Criando o elemento (Aviao) a ser inserido.
novo_dado = Aviao(nome, prioridade)

# Inserindo o novo elemento na posição correta.
self.dados[i + 1] = novo_dado

# Incrementando em uma unidade o valor quantidade.
self.quantidade += 1

return True
```

Metodologia

4. Vetor Ordenado - Remover

Remove o elemento de menor valor (maior prioridade):



Metodologia

4. Vetor Ordenado - Remover

Remove o elemento de menor valor (maior prioridade):

Metodologia

4. Vetor Ordenado - Remover

Remove o elemento de menor valor (maior prioridade):

Complexidade: O(1)

Metodologia

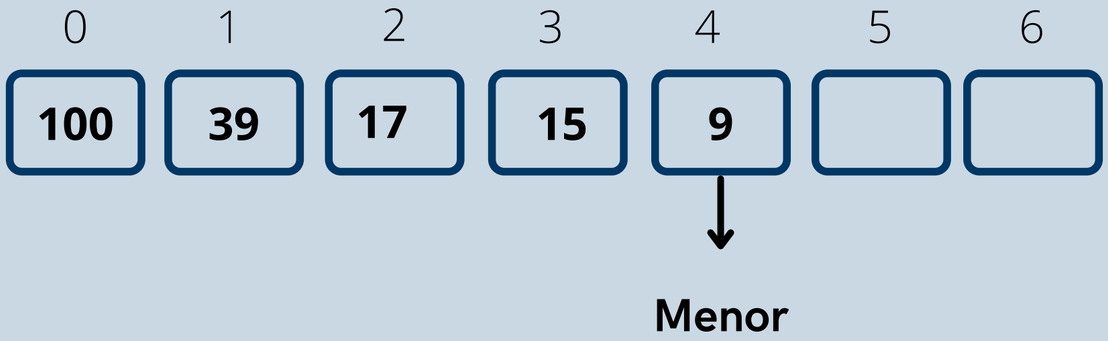
4. Vetor Ordenado - Remover

```
def remover(self):
    # Verificando se a fila está vazia.
    if self.vazia():
      print("Fila Vazia.")
      return False

     # Decrementando em uma unidade o valor quantidade.
    self.quantidade -= 1

    return True
```

Metodologia

4. Vetor Ordenado - Consultar

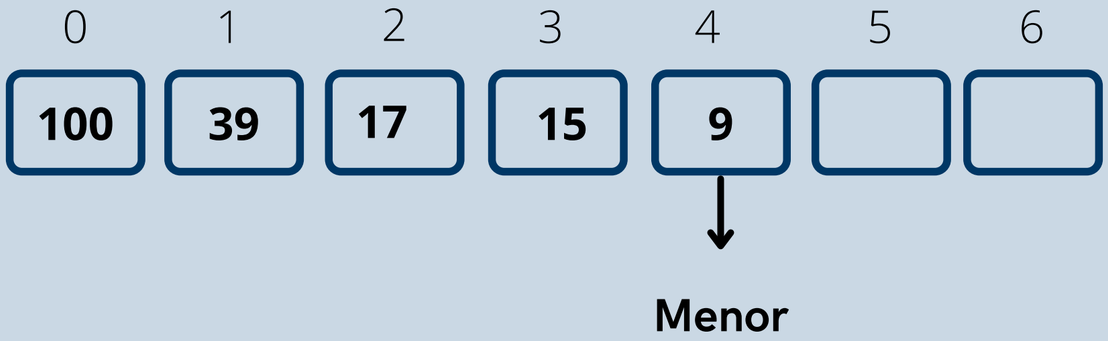
Consultando o elemento de menor valor (maior prioridade):



Metodologia

4. Vetor Ordenado - Consultar

Consultando o elemento de menor valor (maior prioridade):



Complexidade: O(1)

Metodologia

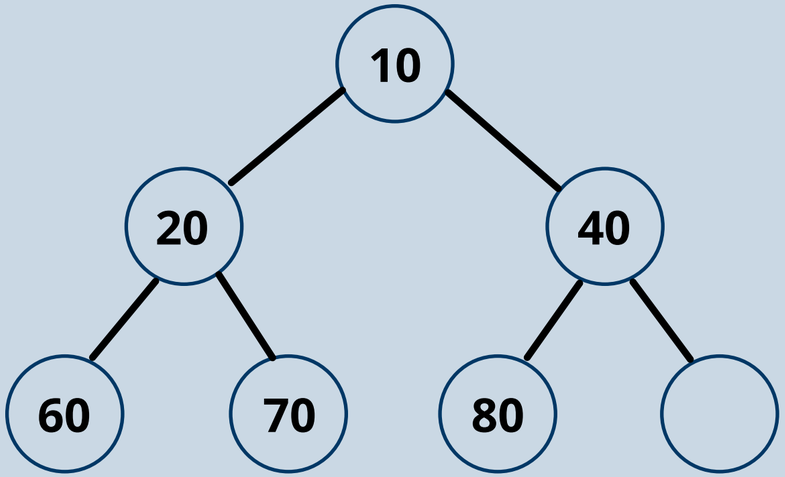
4. Vetor Ordenado - Consultar

```
def consultar(self):
    # Verificando se a fila está vazia.
    if self.vazia():
      print("Fila Vazia.")
      return False

    #Mostrando ultimo elemento do array (maior prioridade)
    print(self.dados[self.quantidade - 1])

    return True
```

Metodologia

5. Heap Binária - Inserir

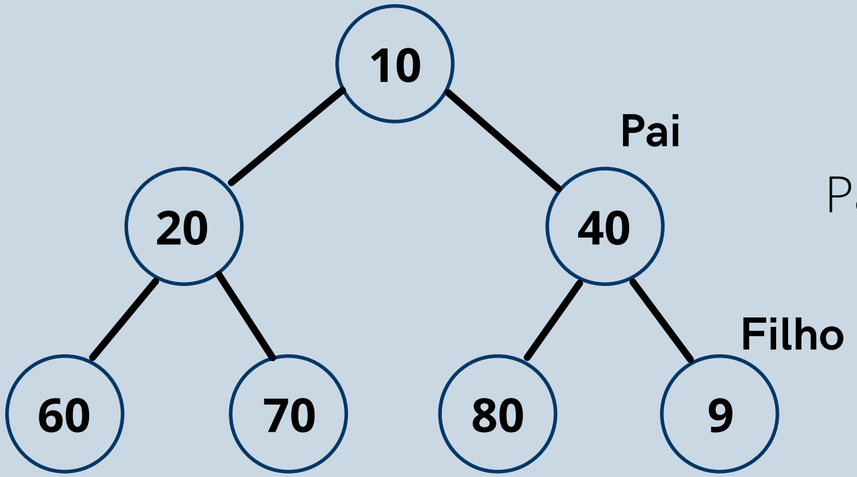
Inserindo o elemento de prioridade 9:



Metodologia

5. Heap Binária - Inserir

Inserindo o elemento de prioridade 9:

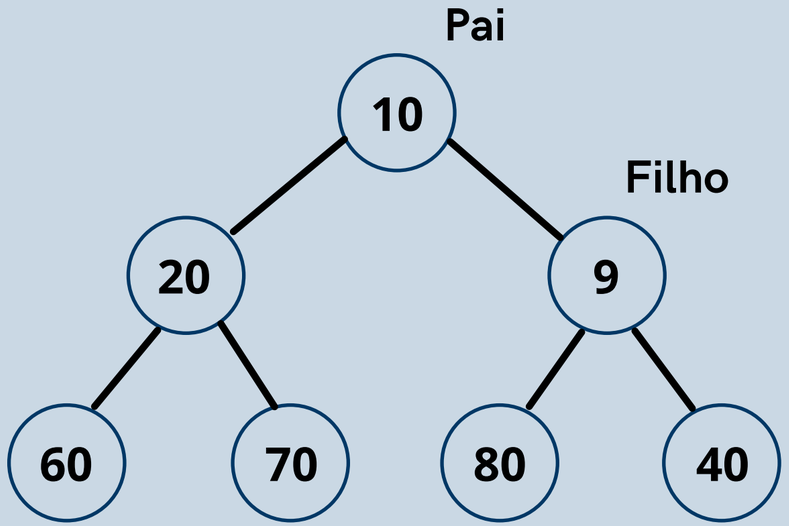


Pai > Filho ⟶ Trocar

Metodologia

5. Heap Binária - Inserir

Inserindo o elemento de prioridade 9:

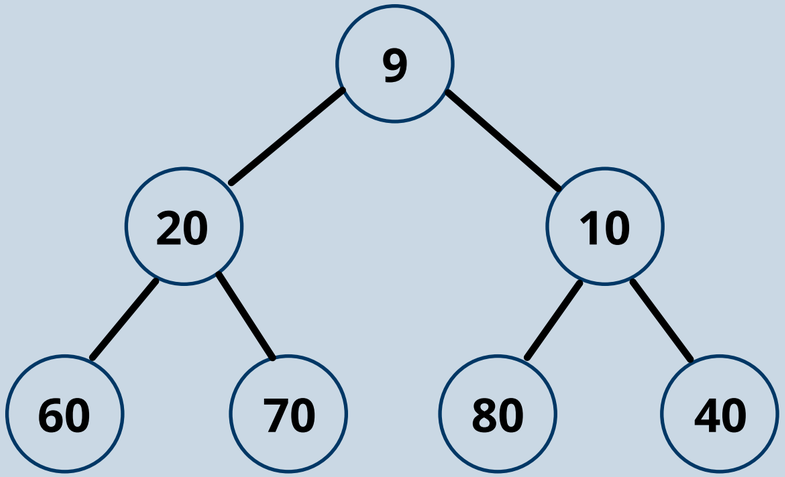


Pai > Filho ⟶ Trocar

Metodologia

5. Heap Binária - Inserir

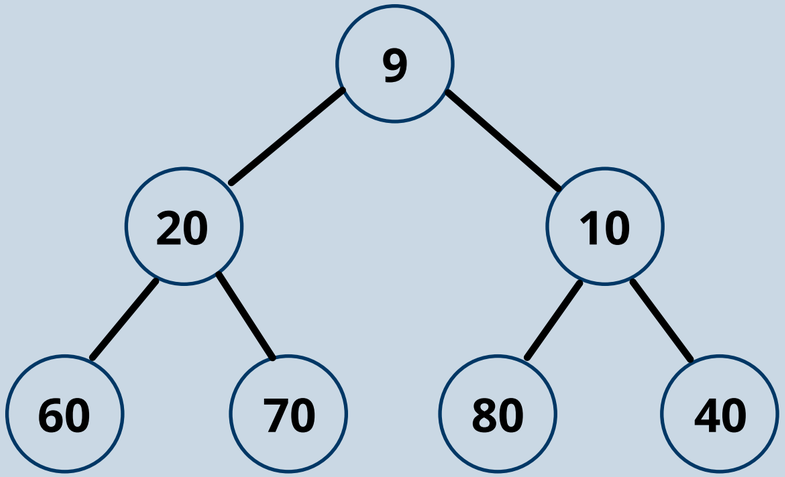
Inserindo o elemento de prioridade 9:



Metodologia

5. Heap Binária - Inserir

Inserindo o elemento de prioridade 9:



Complexidade: O(log n)

Metodologia

5. Heap Binária - Inserir

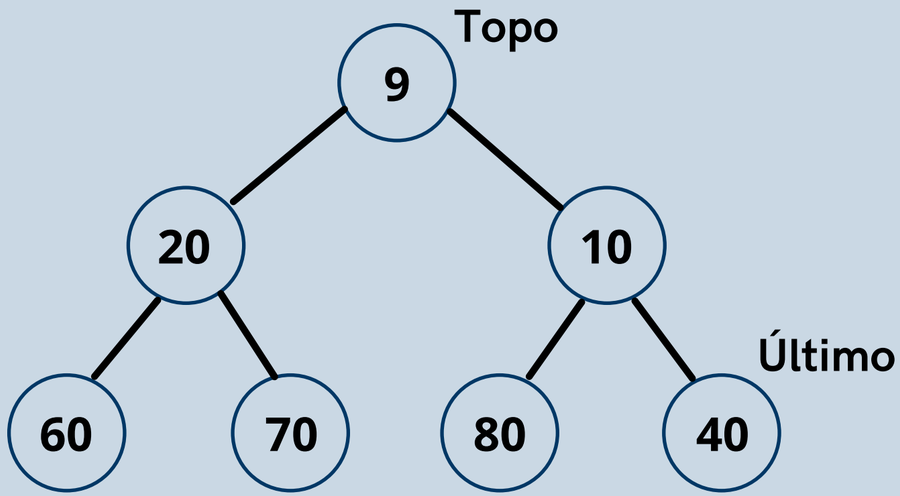
```
def inserir(self, nome, prioridade):
# Verificar se a Heap está cheia.
if self.cheia():
print("Fila Cheia.")
return False

# Criando o elemento (Aviao) a ser inserido.
novo_dado = Aviao(nome, prioridade)

# Inserindo o elemento na ultima posição da Heap.
self.dados[self.quantidade] = novo_dado
```

```
# Subindo o elemento para posição correta.
self.__subir(filho = self.quantidade)

# Incrementando em uma unidade o valor quantidade.
self.quantidade += 1

return True
```

Metodologia

5. Heap Binária - Remover

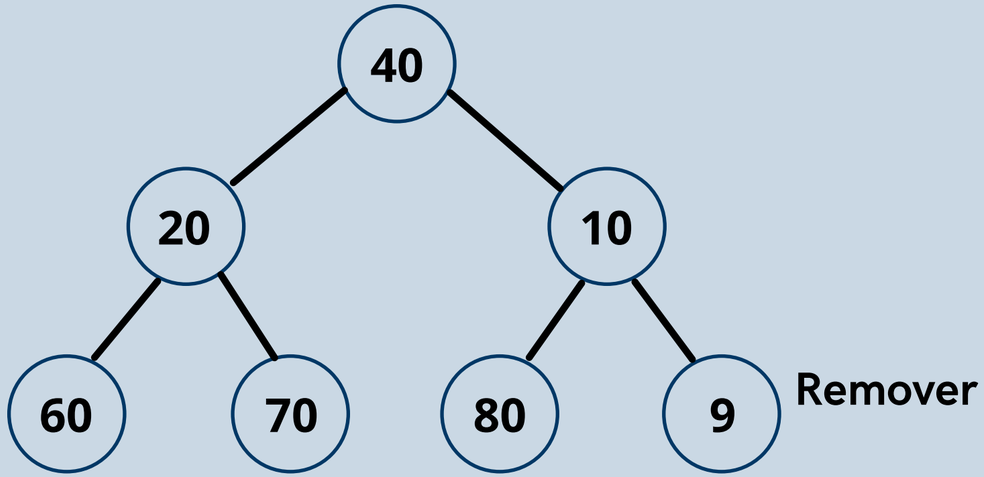
Remove o elemento de menor valor (maior prioridade):



Metodologia

5. Heap Binária - Remover

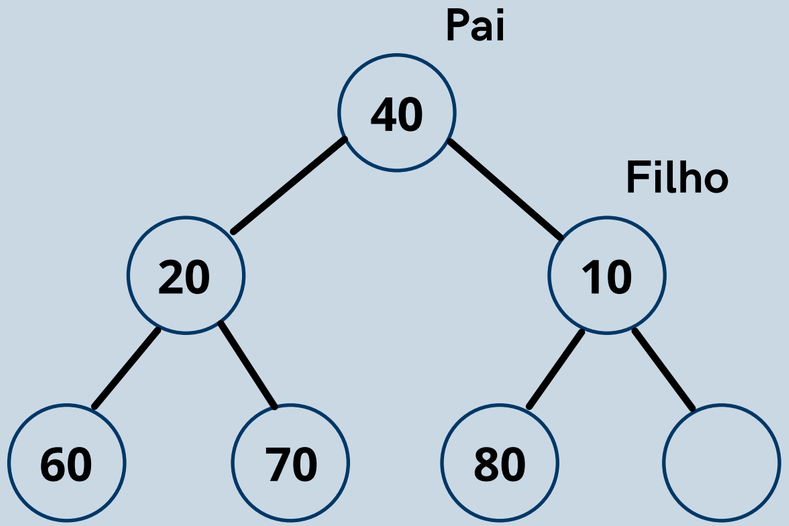
Remove o elemento de menor valor (maior prioridade):



Metodologia

5. Heap Binária - Remover

Remove o elemento de menor valor (maior prioridade):

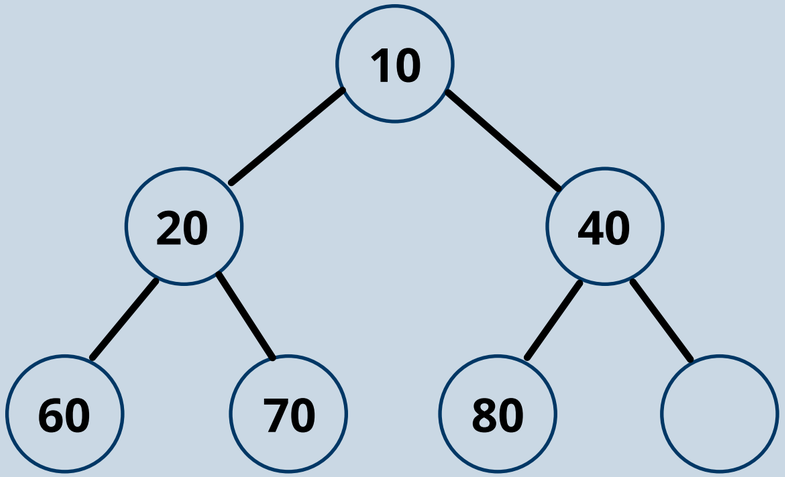


Pai > Filho → Trocar

Metodologia

5. Heap Binária - Remover

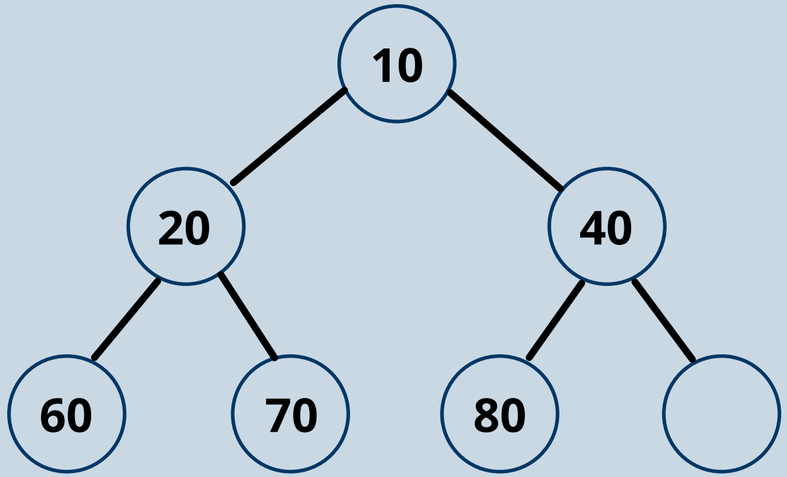
Remove o elemento de menor valor (maior prioridade):



Metodologia

5. Heap Binária - Remover

Remove o elemento de menor valor (maior prioridade):



Complexidade: O(log n)

Metodologia

5. Heap Binária - Remover

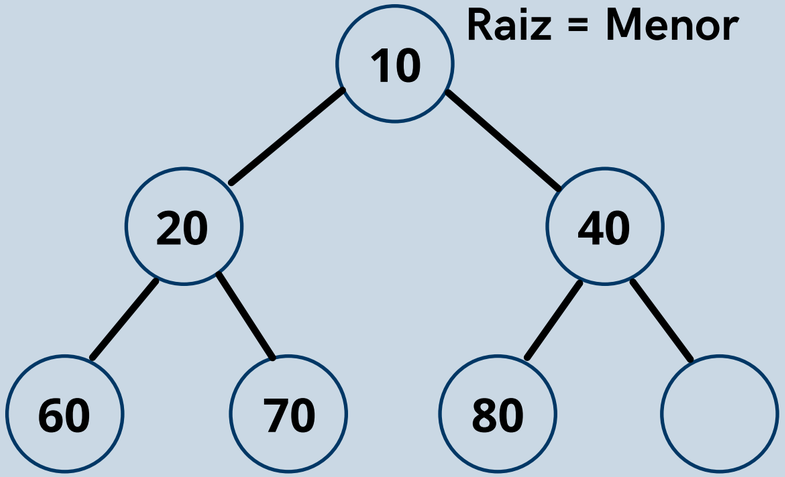
```
def remover(self):
# Verificando se a fila está vazia.
if self.vazia():
print("Fila Vazia.")
return False

# Decrementando em uma unidade o valor quantidade.
self.quantidade -= 1

# Colocando o ultimo elemento no topo (no lugar do elemento removido).
self.dados[0] = self.dados[self.quantidade]
```

```
# Descendo o elemento que foi inserido no topo para a sua posição correta.
self.__descer(pai = 0)

return True
```

Metodologia

5. Heap Binária - Consultar

Consultando o elemento de menor valor (maior prioridade):



Complexidade: O(1)

Metodologia

5. Heap Binária - Consultar

```
def consultar(self):
# Verificando se a fila está vazia.
if self.vazia():
print("Fila Vazia.")
return False

# Mostrando o elemento do topo da fila (maior prioridade).
print(self.dados[0])
```

Metodologia

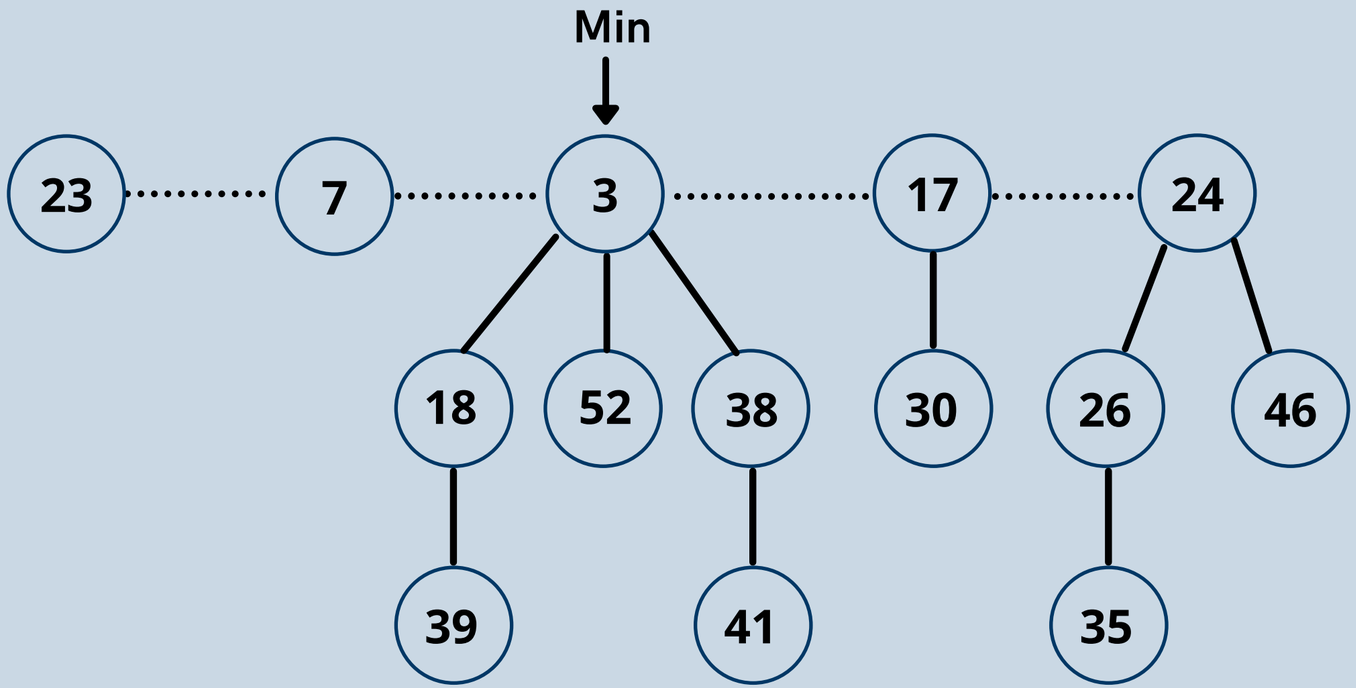
6. Heap de Fibonacci - Aviao Modificada

```
class Aviao:
  def __init__(self) -> None:
    self.nome = None
    self.prioridade = -1
    self.pai = None
    self.filho = None
    self.esquerda = None
    self.direita = None
    self.grau = -1

  def __repr__(self):
     rep = "Nome: " + str(self.nome) + " | " + "Nível de
 Combustível: " + str(self.prioridade)
    return rep

  def __str__(self) -> str:
  return self.__repr__()
```

Metodologia

6. Heap de Fibonacci - Inserir

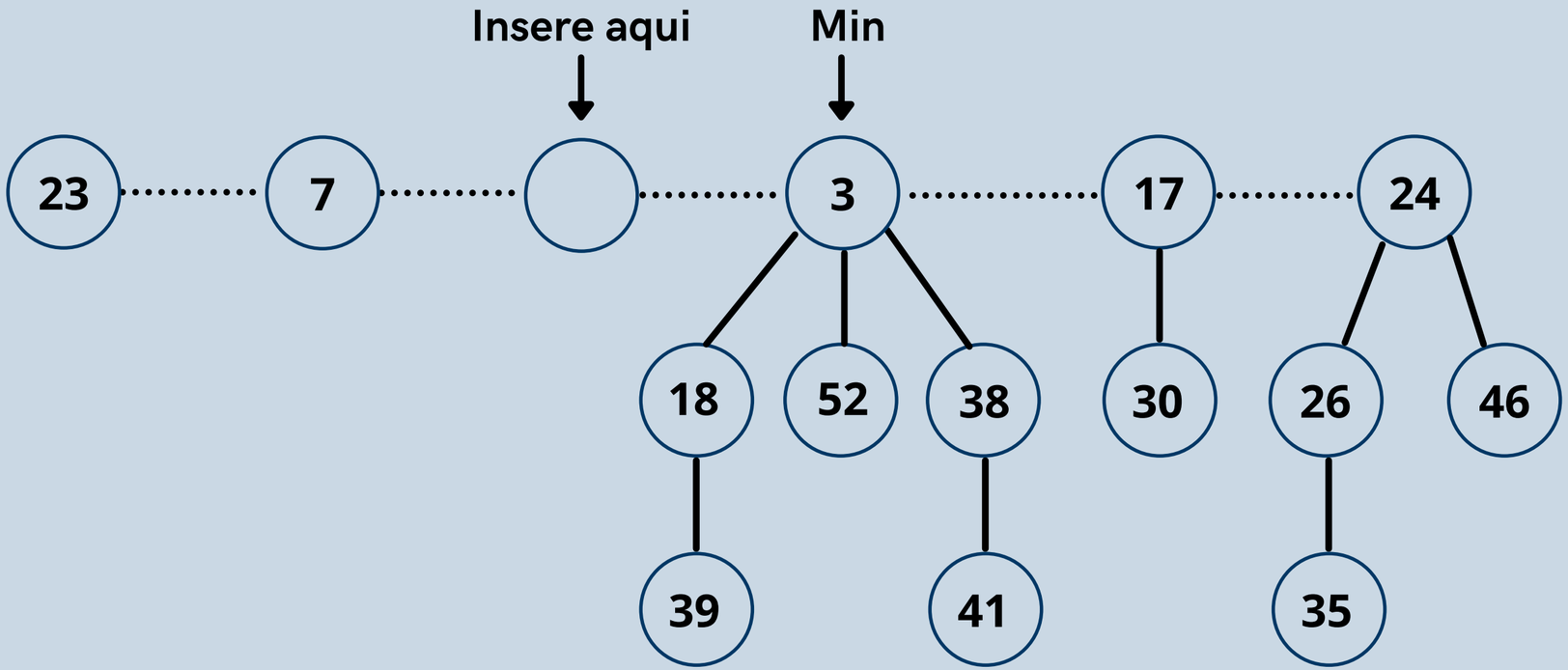
Inserindo o elemento de prioridade 21:



Metodologia

6. Heap de Fibonacci - Inserir

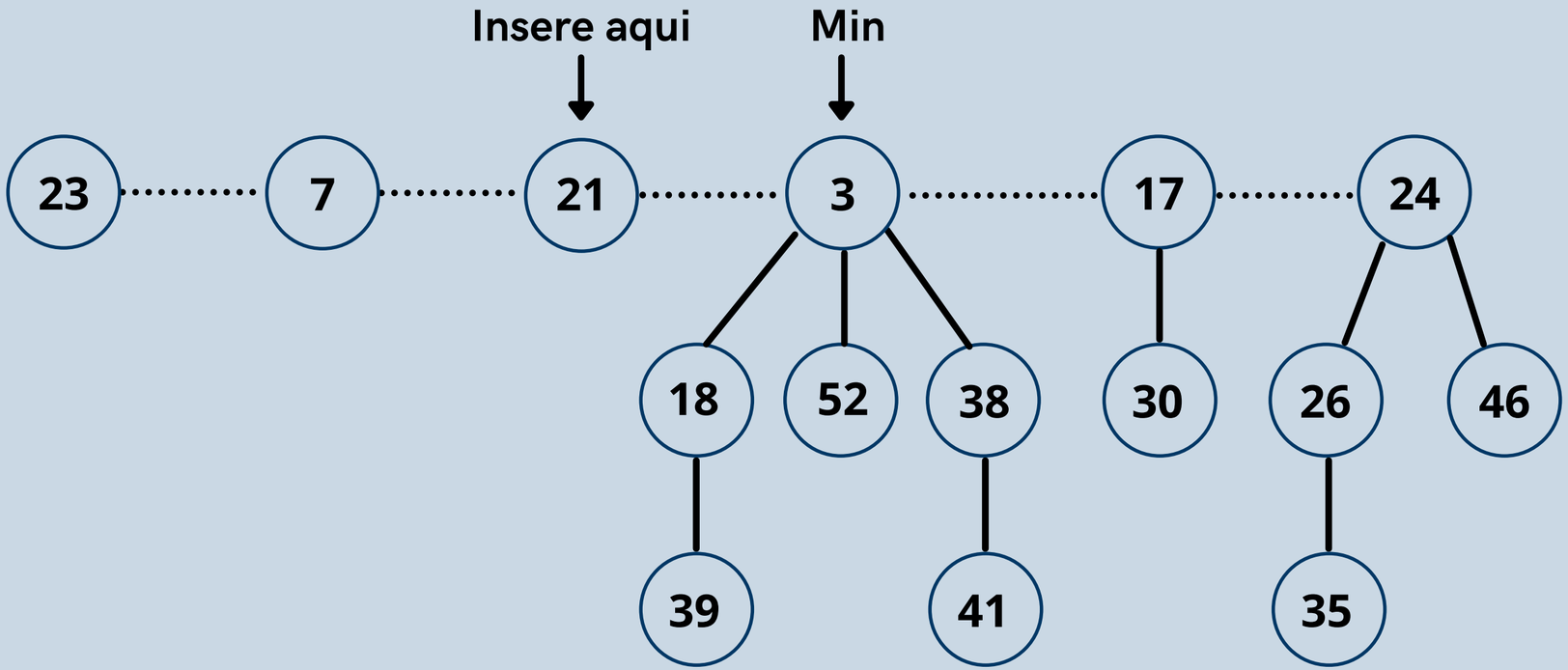
Inserindo o elemento de prioridade 21:



Metodologia

6. Heap de Fibonacci - Inserir

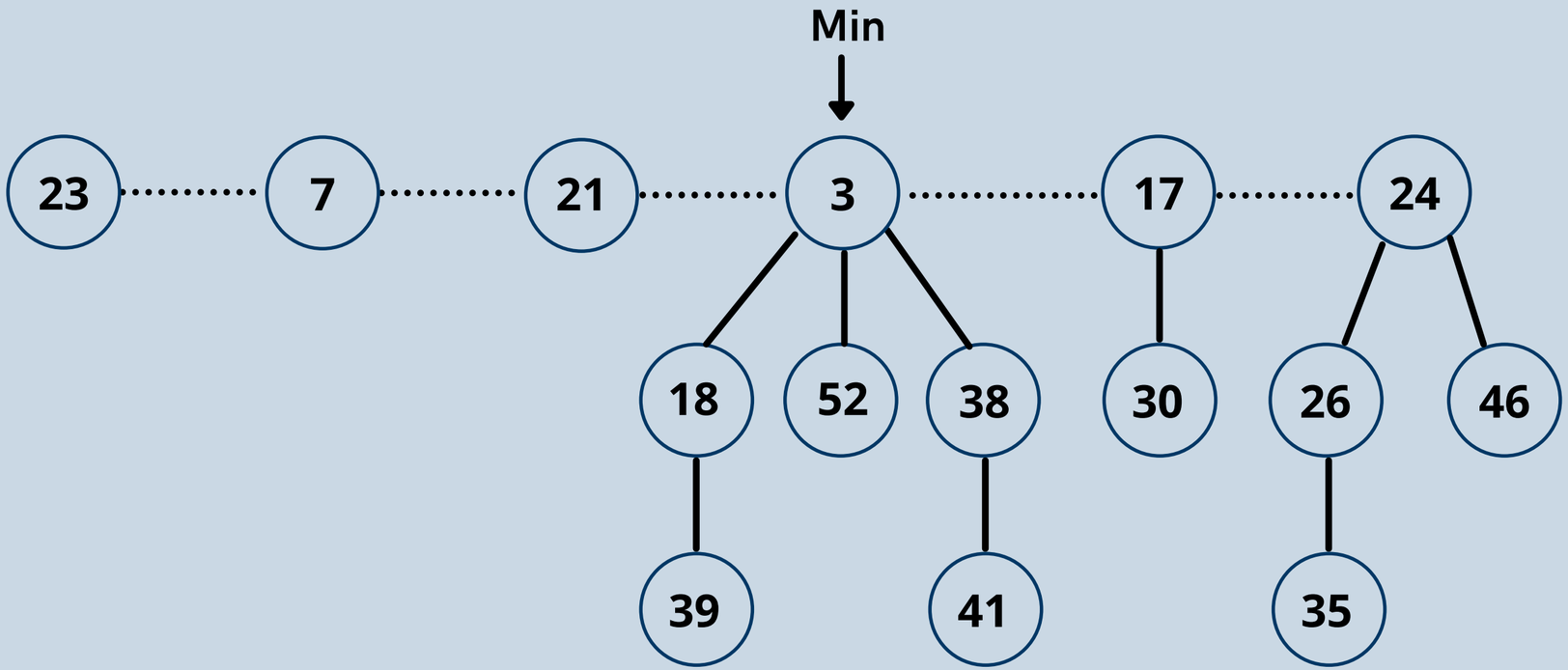
Inserindo o elemento de prioridade 21:



Metodologia

6. Heap de Fibonacci - Inserir

Inserindo o elemento de prioridade 21:



Complexidade: O(1)

Fonte: https://www.programiz.com/dsa/fibonacci-heap

Metodologia

6. Heap de Fibonacci - Inserir

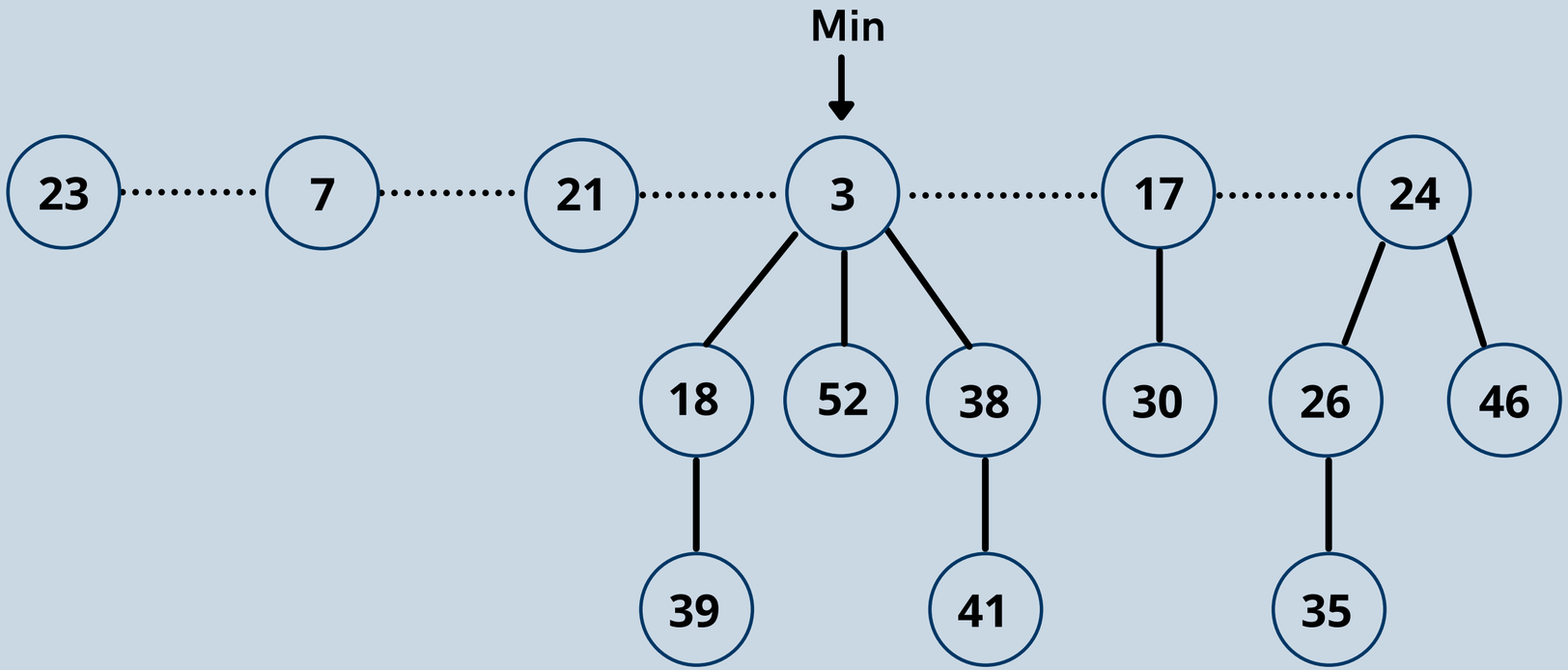
```
def inserir(self, nome, prioridade):
# Criando o elemento (objeto) a ser inserido.
novo_dado = Aviao()

# Inserindo as informações do novo elemento.
novo_dado.nome = nome
novo_dado.prioridade = prioridade
novo_dado.pai = None
novo_dado.filho = None
novo_dado.esquerda = novo_dado
novo_dado.direita = novo_dado
novo_dado.grau = 0
```

```
if self.mini != None:
self.mini.esquerda.direita = novo_dado
novo_dado.direita = self.mini
novo_dado.esquerda = self.mini.esquerda
self.mini.esquerda = novo_dado

if novo_dado.prioridade < self.mini.prioridade:
self.mini = novo_dado
else:
self.mini = novo_dado

self.quantidade += 1
```

Metodologia

6. Heap de Fibonacci - Remover

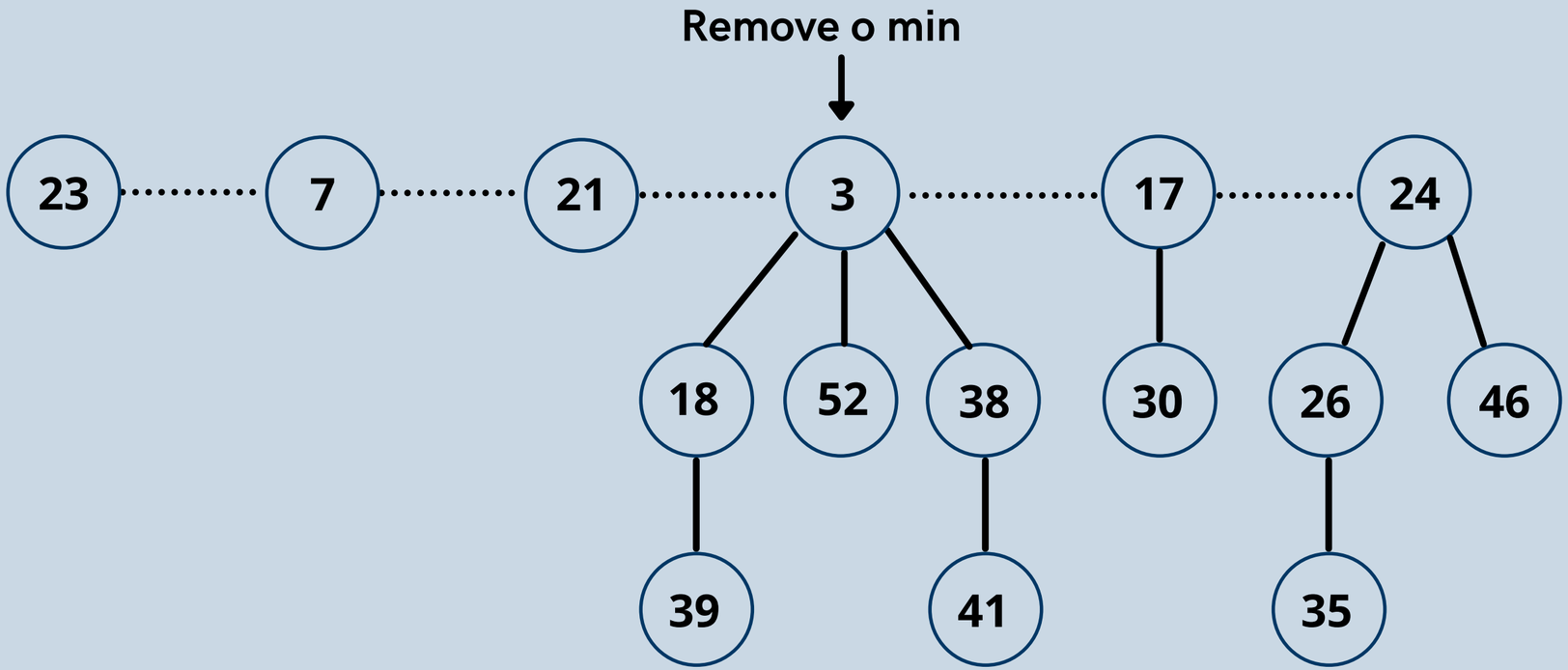
Remove o elemento de menor valor (maior prioridade):



Metodologia

6. Heap de Fibonacci - Remover

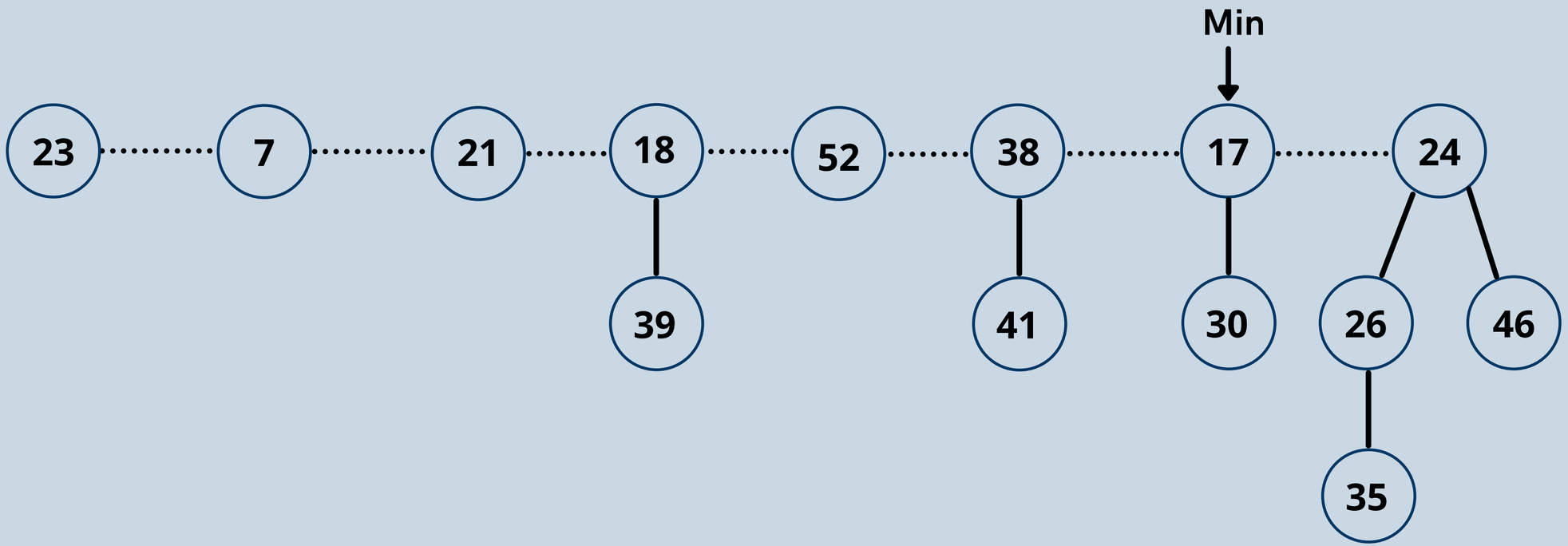
Remove o elemento de menor valor (maior prioridade):



Metodologia

6. Heap de Fibonacci - Remover

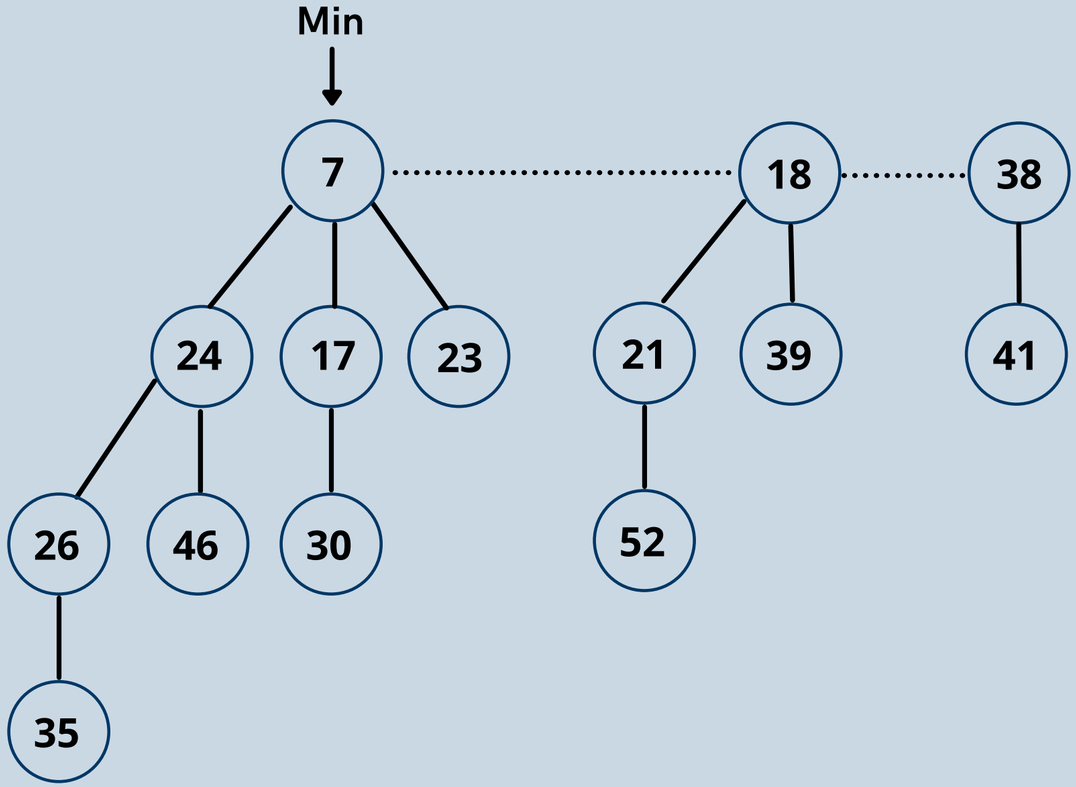
Remove o elemento de menor valor (maior prioridade):



Metodologia

6. Heap de Fibonacci - Remover

Remove o elemento de menor valor (maior prioridade):



Complexidade: O(log n)

Fonte: https://www.programiz.com/dsa/fibonacci-heap

Metodologia

6. Heap de Fibonacci - Remover

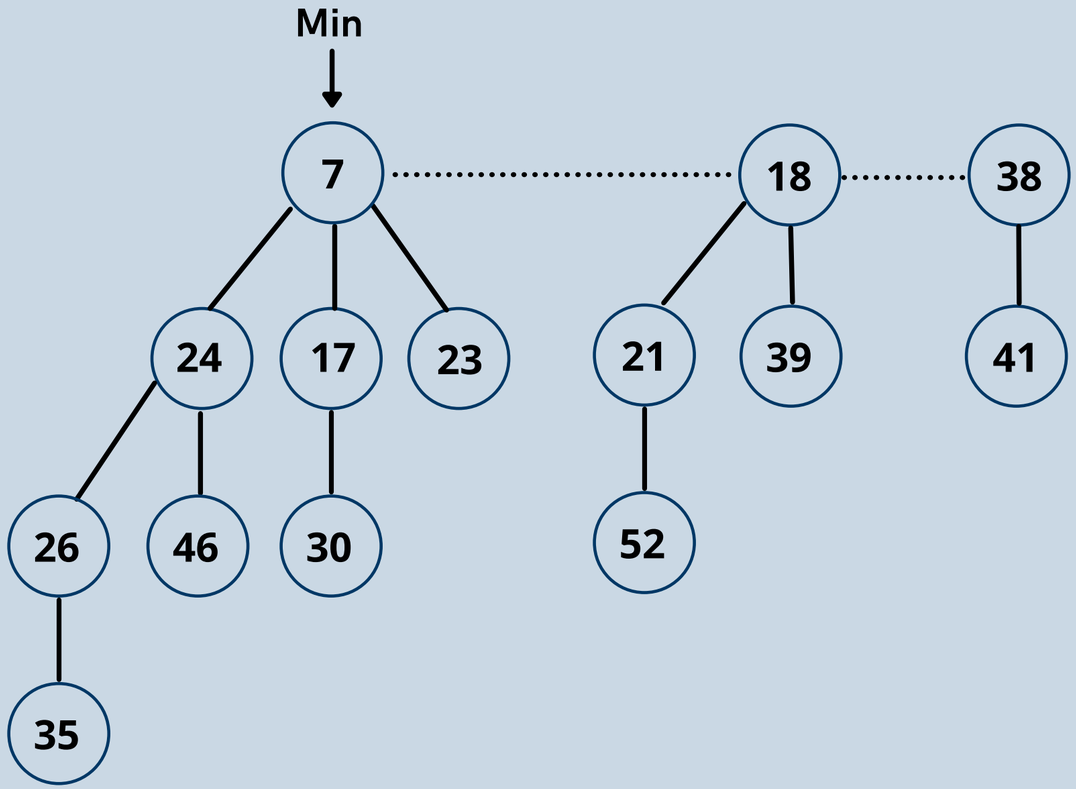
```
def remover(self):
# Verificando se a fila está vazia.
if self.vazia():
print("Fila Vazia.")
return False
else:
temp = self.mini
no_atual = temp
x = None

if temp != None:
if (temp.filho != None):
x = temp.filho

while(True):
no_atual = x.direita
self.mini.esquerda.direita = x
```

```
x.direita = self.mini
x.esquerda = self.mini.esquerda
self.mini.esquerda = x

if x.prioridade < self.mini.prioridade:
self.mini = x

x.pai = None
x = no_atual

if (no_atual == temp.filho):
break

temp.esquerda.direita = temp.direita
temp.direita.esquerda = temp.esquerda
self.mini = temp.direita
```

```
if temp == temp.direita and temp.filho == None:
self.mini = None
else:
self.mini = temp.direita
self.__consolidar()

self.quantidade -= 1
```

Metodologia

6. Heap de Fibonacci - Consulta



Complexidade: O(1)

Fonte: https://www.programiz.com/dsa/fibonacci-heap

Metodologia

6. Heap de Fibonacci - Consultar

```
def consultar(self):
# Verificando se a fila está vazia.
if self.vazia():
print("Fila Vazia.")
return False

# Mostrando o elemento do topo da fila (maior prioridade).
print(self.mini)
```

Metodologia

7. Complexidades dos Algoritmos

Algoritmos	Inserir	Remover	Consultar
Vetor Ordenado	O(1)	O(n)	O(n)
Vetor Desordenado	O(n)	O(1)	O(1)
Heap Binária	O(log n)	O(log n)	O(1)
Heap de Fibonacci	O(1)	O(log n)	O(1)

Tabela 2. Análise assintótica dos algoritmos

Metodologia

8. Configuração da máquina (Testes):

- **Processador:** Ryzen 5 3400G
- **Memória:** 16GB DDR4
- **Sistema Operacional:** Windows 10
- **Linguagem de Programação:** Python 3.9.7
- **Compilador / IDE:** Visual Studio Code 1.63.2

Metodologia

9. Testes e Operações

- Primeiro teste, foram feitos 10.000 vezes (com média aritmética), com 20 inserções, 20 remoções e 20 consultas, obtendo os seguintes tempos:

- Segundo teste, foram feitos 10 vezes (com média aritmética), com 20.000 inserções, 20.000 remoções e 20.000 consultas, obtendo os seguintes tempos:

Conclusão

Tempos de compilação obtidos

Algoritmo	Tempo
Vetor Deordenado	0.003 s
Vetor Ordenado	0.003 s
Heap Binária	0.003 s
Heap de Fibonacci	0.003 s

Tabela 3. Quadro do primeiro teste

Algoritmo	Tempo
Vetor Deordenado	61.877 s
Vetor Ordenado	28.427 s
Heap Binária	3.015 s
Heap de Fibonacci	2.815 s

Tabela 4. Quadro do segundo teste

Conclusão

Levando em consideração as informações obtidas, podemos tirar as seguintes conclusões: No primeiro caso de teste, com apenas vinte operações, o tempo de execução dos algorítmos pouco se diferenciam; entretanto no segundo caso de teste, é perceptível a disparidade entre o tempo de compilação das estruturas.

Deste modo, apesar do Heap de Fibonacci ter sido um algorítmo de destaque com o teste com grandes números de operações e ter si mantido igual às demais estruturas no teste com poucas operações, devemos levar em consideração sobre a aplicação do estudo, a simulação de aeroporto. Neste contexto, não é comum que haja um tráfego aéreo cuja quantidade de aviões seja tão exuberante, dessa forma, apesar do Heap de Fibonacci e Heap Binário terem tido bons resultados, essa situação não condiz com a vida real.

O caso de teste com apenas 20 operações, ou seja, uma circunstância mais comum, manifesta que todas estruturas apresentadas para a simulação de tráfego aéreo possuem uma boa eficácia, ficando apenas a críterio do programador qual algorítmo será utilizado.

Referências Bibliográficas

[1] Fibonacci heap. https://www.growingwiththeweb.com/data-structures/fibonacci-heap/overview/. Acessado em: 09-01-2022.

[2] Fibonacci heap-deletion, extract min and decrease key. FibonacciHeap-Deletion, Extract min and Decreasekey-GeeksforGeeks. Acessado em: 04-01-2022.

[3] Fila de prioridade. http://www.facom.ufu.br/˜abdala/DAS5102/TEO_HeapFilaDePrioridade.pdf. Acessado em: 04-01-2022.

[4] Filas de prioridade e heap. https://www.ic.unicamp.br/˜rafael/cursos/2s2018/mc202/slides/unidade21-fila-de-prioridade.pdf. Acessado em: 04-01-2022.

Referências Bibliográficas

[5] Andre Backes. Estrutura de Dados Descomplicada - em Linguagem C. 2016.

[6] Lilian Markenzon Jayme L. Szwarcfiter. Estruturas de Dados e Seus Algoritmos. LTC, 1994.

[7] Ronald L. Rivest Clifford Stein Thomas H. Cormen, Charles E. Leiserson. Algoritmos - Teoria e Prtica. LTC, 1989.

[9] Mark A. Weiss. Data Structures and Algorithm Analysis in C++. 2006.

Obrigado!