Análise das Estruturas de Prioridade Aplicadas ao Controle de Tráfego Aéreo

Carlos E. Neto, Crislane C. da Silva, Davi C., Douglas A., Wilgner G. Vidal.

[1] Universidade Estadual do Ceará (UECE)
Av. Dr. Silas Munguba – 1700 – Fortaleza – CE – Brasil

[2]Centro de Ciência e Tecnologia (CCT) Fortaleza, CE

[3]Curso de Graduação em Ciência da Computação
Universidade Estadual do Ceará (UECE) – Fortaleza, CE – Brasil

Resumo. *Neste presente trabalho, utilizando a linguagem Python, serão implementados diversos algorítmos de prioridade que seram aplicadas em uma simulação de aeroporto, cujo o papel destas estruturas será de controlar o fluxo aéreo dos aviões baseado na quantidade de gasolina que possuem.*
Palavras-chave: Estrutura de dados. Estruturas de prioridade. Heaps. Controle de tráfego aéreo.

Abstract. *In this present work, utilizing the Python language, will be implemented sundry priority algorithms that will be applied to an airport simulation, whose the role of those structures will be to control the air traffic based on the quantity of gasoline.*
Key-words: Data structures. Priority structures. Heaps. Air traffic control.

1. Introdução

Em inúmeras situações, somos obrigados a determinar a precedência de nossas atitudes. Com isso, neste estudo, temos o objetivo de proporcionar uma compreensão acerca dos algorítmos de prioridade, de forma que seja apresentada sua parte teórico-conceitual em uma aplicação da vida real.

Além disso, este artigo foi produzido com o intuito de explorar os conceitos das estruturas de prioridade em situações que possuem a necessidade de um controle de precedências, sendo importante identificar qual algorítmo possui maior eficiência.

2. Fundamentação Teórica

2.1. Estruturas de Dados

Antes de comentar sobre os algoritmos de prioridade, é interessante fazer uma breve contextualização sobre o que são estruturas de dados: Estruturas de dados são formas específicas de organizar os dados, geralmente feitas para que os dados possam ser administrados de maneira mais efetiva. De modo geral, as estruturas de dados fornecem

um meio de gerenciar grandes quantidades de elementos com eficiência para usos como grandes bancos de dados.

As estruturas de dados servem como base para os tipos abstratos de dados (TAD), estes que determinam a lógica por de trás da estrutura de dados. Deste modo, o TAD funciona para declarar as operações da estrutura de dados, mas não necessariamente mostrar como as operações estão sendo implementadas, assim originando o conceito de abstração.

Desta forma, determinadas estruturas de dados servem para específicas situações, como por exemplo: Tabelas Hash funcionam para implementação de compiladores; Árvores Splay para consultas bancárias; e Fila de Prioridade para ordem de atendimento em um hospital.

2.2. Fila de Prioridade

A priori, uma fila consiste em uma estrutura de dados linear que funciona de maneira FIFO, *First In First Out*, ou seja, o primeiro elemento a entrar na fila, será o primeiro a sair.

Neste contexto, a fila de prioridade consiste em um tipo especial de fila, na qual possui um conjunto de itens dispostos em uma ordem de acordo com as prioridades de cada elemento, ou seja, é esse valor de prioridade que determinará a posição que um novo elemento deve ser inserido na fila. Da mesma forma, o valor da prioridade determinará também qual elemento será removido ao efetuar a operação de remoção. Se tratando de fila de prioridade o elemento a ser removido será o de maior prioridade, conforme a figura 1:

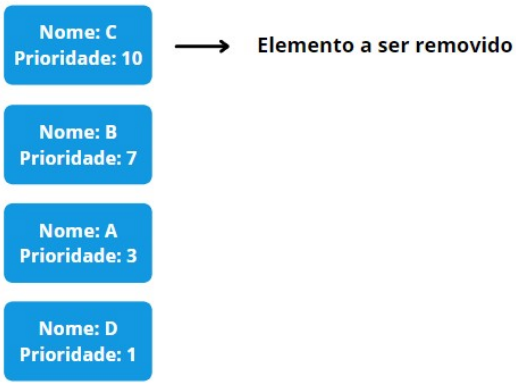

Figura: 1
Fonte: Arquivo pessoal

Na maioria das implementações de estruturas de dados, o usuário não tem acesso a elas de forma direta, tendo somente acesso a operações específicas. De modo geral, a fila possui as seguintes operações:

- Criar a fila;
- Inserção (levando em consideração o valor de prioridade);
- Remoção (levando em consideração o valor da prioridade);
- Consultar um elemento (identificar prioridade) e
- Verificar informações da fila (se está cheia ou vazia).

A partir do conceito desta estrutura, é possível implementar demais algorítmos utilizando fila de prioridade. Dentre elas estão:

- Array desordenado;
- Array ordenado;
- Lista dinâmica encadeada (neste trabalho não usaremos esse tipo);
- Heaps (neste trabalho usaremos apenas a Heap Binária e a Heap de Fibonacci).

2.3. Heap Binário

A partir de uma fila de prioridade, pode ser implementada a Heap Binária, que permite simular uma árvore binária completa, ou quase completa, tendo o seu último nível incompleto e com os seus elementos o mais à esquerda possível. Quando se pensa no funcionamento de uma Heap, cada posição do array passa a ser considerado o pai de duas outras posições, chamadas filhos, seguindo a propriedade na qual a posição i passa a ser o pai das posições, já seus filhos a esquerda 2*i + 1 e seus filhos da direita 2*i + 2. Ademais, todos os elementos estarão dispostos na heap de forma que o pai tenha sempre uma prioridade "maior ou igual" à prioridade de seus filhos.

Além disso, existem dois tipos de heaps: Em um heap de máximo, o maior valor do conjunto está na raiz da árvore, enquanto no heap de mínimo a raiz armazena o menor valor existente.

Segue abaixo operações comuns em uma Heap Binária:

- Inserção (levando em consideração o valor de prioridade);
- Remoção (levando em consideração o valor da prioridade);
- Consultar um elemento (identificar prioridade) e
- Verificar informações da heap (se está cheia ou vazia).

2.4. Heap de Fibonacci

O Heap de Fibonacci surgiu como uma variação do Heap Binomial. Deste modo, o Heap de Fibonacci é uma coleção de árvores que possuem a propriedade de heaps mínimos ou máximos. Neste heap, um nó pode ter mais de dois filhos ou nenhum filho, além disso, o nó com menor valor sempre será apontado por um ponteiro.

Os Heaps de Fibonacci usam listas circulates duplamente encadeadas para permitir o tempo O(1) para operações como separar uma parte de uma lista, concatenar duas listas e encontrar o valor mínimo ou máximo. Ademais, os nós filhos de um nó pai são conectados uns aos outros por meio de uma lista circular duplamente encadeada, assim como os nós pais estão ligados aos filhos.

De modo geral, o Heap de Fibonacci possui as seguintes operações:

- Inserção (levando em consideração o valor de prioridade);
- Remoção (levando em consideração o valor de prioridade);
- Encotrar mínimo (retornar o valor mínimo) e
- União (concatenar a heap de Fibonacci com alguma outra estrutura válida)

3. Metodologia

3.1. Problema e solução

Neste presente estudo, abordamos o problema de 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.

Este trabalho 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 nivel de combustível terá a prioridade para pouso, com intuito de evitar acidentes por falta de abastecimento. Semelhante ao esquema abaixo:

Tabela 1. Quadro de prioridades de pousos:

Prioridade	Nome	Nivel 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%

3.2. Testes e Operações

Para todos os algorítmos, consistiram de dois testes, um com 20.000 (vinte mil) operações de inserção, remoção e consulta; e o outro com 20 (vinte) operações de inserção, remoção e consulta. Além disso, todas as estruturas foram executadas 10 vezes e em seguida feito a média aritimética do tempo de compilação. Além disso, no primeiro caso de teste, as estruturas foram executadas dez vezes e em seguida feita a média aritimética do tempo de compilação, enquanto o segundo teste foi feita a média aritimética de dez mil tempos de compilação.

3.3. Implementações

Nesse trabalho utilizou-se a linguagem Python, como linguagem de programação para implementar os algoritmos citados anteriormente. Inicialmente, foi criado uma classe Avião, que representam os aviões que se encontram na fila de prioridade. A classe Aviao será utilizada em todos os algoritmos que veremos posteriormente. Segue a sua implementação abaixo:

```
1  class Aviao():
2      def __init__(self, nome = None, prioridade = None) -> None:
3          self.nome = nome
4          self.prioridade = prioridade
5
6      def __repr__(self):
7          rep = "Nome: " + str(self.nome) + " | " + "Nivel de Combustivel
               : " + str(self.prioridade) + "%"
8          return rep
9
10     def __str__(self) -> str:
11         return self.__repr__()
```

3.3.1. Array Desordenado

A classe FP_VetorDesordenado é a classe que representa a fila de prioridades implementada com o array desordenado. Segue a sua implementação abaixo:

```
1  from Aviao import Aviao
2  import time
3
4  class FP_VetorDesordenado:
5      def __init__(self, tamanho_maximo) -> None:
6          self.quantidade = 0
7          self.tamanho_maximo = tamanho_maximo
8          self.dados = [None] * tamanho_maximo
```

No array desordenado a função inserir posiciona um novo elemento diretamente na última posição livre do array, sem que haja preocupação com a ordem de prioridade, logo, temos uma complexidade *O(1)*. Segue a implementação da função inserir abaixo:

```
1      def inserir(self, nome, prioridade):
2          # Verificando se a fila está cheia.
3          if self.cheia():
4              print("Fila Cheia.")
5              return False
6
7          # Criando o elemento (Aviao) a ser inserido.
8          novo_dado = Aviao(nome, prioridade)
9
10         # Inserindo o novo elemento na última posição.
11         self.dados[self.quantidade] = novo_dado
12
13         # Incrementando em uma unidade o valor quantidade.
14         self.quantidade += 1
15
16         return True
```

Na função remover do array desordenado é necessário percorrer o array para encontrar o elemento de maior prioridade (o avião com menor nivel de combustível) para ser removido, levando no pior caso a complexidade *O(n)*. Segue a implementação da função remover abaixo:

```
1      def remover(self):
2          # Verificando se a fila está vazia.
3          if self.vazia():
4              print("Fila Vazia.")
5              return False
6
7          # Procurando o elemento de menor prioridade.
8          menor = self.dados[0]
9          indice = 0
10
11         for i in range(1, self.quantidade):
12             if menor.prioridade > self.dados[i].prioridade:
13                 menor = self.dados[i]
14                 indice = i
15
```

```
16          # Preenchendo o elemento removido com seus sucessores.
17          for i in range(indice, self.quantidade - 1):
18              self.dados[i] = self.dados[i + 1]
19
20          # Decrementando em uma unidade o valor quantidade.
21          self.quantidade -= 1
22
23          return True
```

Da mesma forma que a função remoção, na função consultar é necessário percorrer o array da fila de prioridade para encontrar o avião que tem maior prioridade, possuindo também, uma complexidade *O(n)*. Segue a implementação da função consultar abaixo:

```
1       def consultar(self):
2           # Verificando se a fila está vazia.
3           if self.vazia():
4               print("Fila Vazia.")
5               return False
6
7           # Procurando o elemento de maior prioridade.
8           menor = self.dados[0]
9           indice = 0
10
11          for i in range(1, self.quantidade):
12              if menor.prioridade > self.dados[i].prioridade:
13                  menor = self.dados[i]
14                  indice = i
15
16          # Mostrando o elemento de maior prioridade
17          print(self.dados[indice])
18
19          return True
```

3.3.2. Array Ordenado

A classe FP_VetorOrdenado que representa a implementação da a fila de prioridades executada com o array ordenado. Segue o código abaixo:

```
1   from Aviao import Aviao
2   import time
3
4   class FP_VetorOrdenado:
5       def __init__(self, tamanho_maximo) -> None:
6           self.quantidade = 0
7           self.tamanho_maximo = tamanho_maximo
8           self.dados = [None] * tamanho_maximo
```

A implementação abaixo mostra como foi feita a função de inserção. Neste caso, o novo elemento é diretamente posicionado no local exato onde equivale sua prioridade, tendo assim a complexidade *O(n)*.

```
1       def inserir(self, nome, prioridade):
2           # Verificando se a fila está cheia.
```

```
3           if self.cheia():
4               print("Fila Cheia.")
5               return False
6
7           # Criando variável para iterar no loop.
8           i = self.quantidade - 1
9
10          # Liberando a posição correta para inserir o novo elemento.
11          while i >= 0 and self.dados[i].prioridade <= prioridade:
12              self.dados[i + 1] = self.dados[i]
13              i -= 1
14
15          # Criando o elemento (Aviao) a ser inserido.
16          novo_dado = Aviao(nome, prioridade)
17
18          # Inserindo o novo elemento na posição correta.
19          self.dados[i + 1] = novo_dado
20
21          # Incrementando em uma unidade o valor quantidade.
22          self.quantidade += 1
23
24          return True
```

Em seguida, na função remover, basta decrementar o último elemento, pois o array já está ordenado, sendo assim sua complexidade *O(1)*

```
1       def remover(self):
2           # Verificando se a fila está vazia.
3           if self.vazia():
4               print("Fila Vazia.")
5               return False
6
7           # Decrementando em uma unidade o valor quantidade.
8           self.quantidade -= 1
9
10          return True
```

A implementação abaixo mostra como foi feita a função de consultar. Este mostra o último elemento do array, ou seja, o que tem menor prioridade, dessa forma, com a complexidade de *0(1)*

```
1       def consultar(self):
2           # Verificando se a fila está vazia.
3           if self.vazia():
4               print("Fila Vazia.")
5               return False
6
7           # Mostrando ultimo elemento do array (maior prioridade)
8           print(self.dados[self.quantidade - 1])
9
10          return True
```

3.3.3. Heap Binária

A classe FP_HeapBinaria é a implementação que representa a fila de prioridades implementada com o array ordenado. Segue o código abaixo:

```
1  from Aviao import Aviao
2  import time
3
4  class FP_HeapBinaria:
5      def __init__(self, tamanho_maximo) -> None:
6          self.quantidade = 0
7          self.tamanho_maximo = tamanho_maximo
8          self.dados = self.dados = [None] * tamanho_maximo
```

A seguir, para inserir um elemento na heap, a função segue o seguinte processo: Primeiramente, é adicionado o elemento no final da heap, em seguida se utiliza a função auxiliar "__subir" para comparar o valor deste nó folha com seus pais, fazendo com que ele suba na árvore de acordo com sua prioridade. Desta forma, sua complexidade se torna *O(log n)*.

```
1      def inserir(self, nome, prioridade):
2          # Verificar se a Heap está cheia.
3          if self.cheia():
4              print("Fila Cheia.")
5              return False
6
7          # Criando o elemento (Aviao) a ser inserido.
8          novo_dado = Aviao(nome, prioridade)
9
10         # Inserindo o elemento na ultima posição da Heap.
11         self.dados[self.quantidade] = novo_dado
12
13         # Subindo o elemento para posição correta.
14         self.__subir(filho = self.quantidade)
15
16         # Incrementando em uma unidade o valor quantidade.
17         self.quantidade += 1
18
19         return True
```

Na função remover, primeiramente, é substituído a raiz pelo o último elemento da árvore, em seguida, a antiga raiz é deletada, logo depois, é utilizada a função auxiliar "__descer"para comparar o nó raiz com seus filhos e desloca-lo até o local pertencente a sua prioridade, dessa forma, sua complexidade é *O(log n)*.

```
1      def remover(self):
2          # Verificando se a fila está vazia.
3          if self.vazia():
4              print("Fila Vazia.")
5              return False
6
7          # Decrementando em uma unidade o valor quantidade.
8          self.quantidade -= 1
9
```

```
10          # Colocando o ultimo elemento no topo (no lugar do elemento
                removido).
11          self.dados[0] = self.dados[self.quantidade]
12
13          # Descendo o elemento que foi inserido no topo para a sua
                posição correta.
14          self.__descer(pai = 0)
15
16          return True
```

Para consultar um elemento na heap, a função simplesmente retorna o valor do nó raiz, ou seja, o que possui maior prioridade, tendo sua complexidade *O(1)*

```
1       def consultar(self):
2           # Verificando se a fila está vazia.
3           if self.vazia():
4               print("Fila Vazia.")
5               return False
6
7           # Mostrando o elemento do topo da fila (maior prioridade).
8           print(self.dados[0])
```

3.3.4. Heap de Fibonacci

Inicialmente, a classe "Aviao"foi modificada para se adaptar ao Heap de Fibonacci, pois agora representa nós de uma árvore. Segue abaixo sua implementação:

```
1   class Aviao:
2       def __init__(self) -> None:
3           self.nome = None
4           self.prioridade = -1
5           self.pai = None
6           self.filho = None
7           self.esquerda = None
8           self.direita = None
9           self.grau = -1
10
11      def __repr__(self):
12          rep = "Nome: " + str(self.nome) + " | " + "Nível de Combustível
                : " + str(self.prioridade)
13          return rep
14
15      def __str__(self) -> str:
16          return self.__repr__()
```

A classe FP_HeapFibonacci é a implementação que representa a fila de prioridade que foi executada com o array ordenado. Além disso, vale ressaltar que a classe "Aviao"foi alterada para representar os nós de uma árvore.

```
1   from Aviao import Aviao
2   import time, math
3
4   class FP_HeapFibonacci:
```

```
5       def __init__(self) -> None:
6           self.mini = None
7           self.quantidade = 0
```

Para a função de inserção, o novo elemento é adicionado como raiz da árvore, e, caso tenha o valor com maior prioridade, ele receberá o título de ”mínimo”, dessa forma, possui a complexidade *O(1)*

```
1       def inserir(self, nome, prioridade):
2           # Criando o elemento (objeto) a ser inserido.
3           novo_dado = Aviao()
4
5           # Inserindo as informações do novo elemento.
6           novo_dado.nome = nome
7           novo_dado.prioridade = prioridade
8           novo_dado.pai = None
9           novo_dado.filho = None
10          novo_dado.esquerda = novo_dado
11          novo_dado.direita = novo_dado
12          novo_dado.grau = 0
13
14          if self.mini != None:
15              self.mini.esquerda.direita = novo_dado
16              novo_dado.direita = self.mini
17              novo_dado.esquerda = self.mini.esquerda
18              self.mini.esquerda = novo_dado
19
20              if novo_dado.prioridade < self.mini.prioridade:
21                  self.mini = novo_dado
22          else:
23              self.mini = novo_dado
24
25          self.quantidade += 1
```

Na função de remoção do elemento, em primeiro passo, é deletado o título de mínimo e é transferido para o próximo elemento de maior prioridade, em seguida, os nós filhos da raiz são deslocados para a lista raiz. A partir daí, são unificados os nós com mesma profundidade na lista raiz até que não haja mais nós pendentes, e após todo esse processo de concatenação, é realocado o título de ”mínimo”caso necessário. Dessa forma, sua complexidade é *O(log n)*

```
1       def remover(self):
2           # Verificando se a fila está vazia.
3           if self.vazia():
4               print("Fila Vazia.")
5               return False
6           else:
7               temp = self.mini
8               no_atual = temp
9               x = None
10
11              if temp != None:
12                  if (temp.filho != None):
13                      x = temp.filho
```

```
14
15                      while(True):
16                          no_atual = x.direita
17                          self.mini.esquerda.direita = x
18                          x.direita = self.mini
19                          x.esquerda = self.mini.esquerda
20                          self.mini.esquerda = x
21
22                          if x.prioridade < self.mini.prioridade:
23                              self.mini = x
24
25                          x.pai = None
26                          x = no_atual
27
28                          if (no_atual == temp.filho):
29                              break
30
31                  temp.esquerda.direita = temp.direita
32                  temp.direita.esquerda = temp.esquerda
33                  self.mini = temp.direita
34
35                  if temp == temp.direita and temp.filho == None:
36                      self.mini = None
37                  else:
38                      self.mini = temp.direita
39                      self.__consolidar()
40
41                  self.quantidade -= 1
```

Na implementação de consulta, é apenas retornado o valor com maior prioridade, ou seja, o nó com o título de mínimo, tendo assim uma complexidade *O(1)*.

```
1       def consultar(self):
2           # Verificando se a fila está vazia.
3           if self.vazia():
4               print("Fila Vazia.")
5               return False
6
7           # Mostrando o elemento do topo da fila (maior prioridade).
8           print(self.mini)
```

3.3.5. Complexidade dos Algorítmos

A seguir, uma recapitulação das análises assintóticas das estruturas implementadas:

Análise Assintótica				
	Array Desordenado	**Array Ordenado**	**Heap Binária**	**Heap de Fibonacci**
Inserir	*O(1)*	*O(n)*	*O(log n)*	*O(1)*
Remover	*O(n)*	*O(1)*	*O(log n)*	*O(log n)*
Consultar	*O(n)*	*O(1)*	*O(1)*	*O(1)*

4. Conclusão e Considerações Finais

Após ser feito os testes nas estruturas implementadas, foi obtida as seguintes informações: Nos testes de vinte operações, o Vetor Desordenado concluiu as operações em 0.003 segundos; o Vetor Ordenado em 0.003 segundos; o Heap Binário em 0.003 segundos; e o Heap de Fibonacci em 0.003 segundos. Para o segundo caso de teste, foram obtidos os demais resultados: O Vetor Desordenado concluiu as operações em 61.877 segundos; o Vetor Ordenado em 28.427 segundos; o Heap Binário em 2.795 segundos; e o Heap de Fibonacci em 2.650 segundos.

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 cosideração 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.

Outrossim, como final do projeto, foi utilizado o Heap de Fibonacci para criar uma interface gráfica com a ferramenta PyQt5 com o intuito de explorar ainda mais a parte prática do estudo, aproximando mais o problema real na forma de um simulador.

5. Bibliografia

[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, ExtractminandDecreasekey-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.

[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] Karleigh Moore.

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

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