

# Pilhas e Filas

Gustavo Aparecido de Souza Viana

**Abstract**—O conceito de pilha e fila é utilizado em diversos lugares, tanto na computação quanto no dia-a-dia. Portanto, neste trabalho será analisado o comportamento dos algoritmos de pilha e fila estática, ou seja, a complexidade do mesmo, do ponto de vista de estrutura de dados. Apesar de serem bem antigos e simples de serem implementados, é muito utilizado na resolução de problemas. Em todos os casos, foi possível observar que na remoção e inserção (na pilha, *pop* e *push*, e na fila, *queue* e *dequeue*) a complexidade é de ordem  $O(1)$ .

## I. INTRODUÇÃO

As pilhas e filas são algoritmos bem utilizados na computação para resolver problemas relacionados a gerenciamento de dados de uma forma simples e minimalista, ou seja, uma melhor estrutura para fazer inserção e remoção do dado.

Podemos dar o exemplo do *RabbitMQ* para filas, o mesmo utiliza o conceito de fila para gerenciar o processamento de dados, ou seja, podemos enfileirar uma sequência de dados a serem processados em uma ordem definida FIFO (*First-In-First-Out*).

Para pilhas, podemos citar um exemplo que é a própria área de memória estática de nossos computadores, a *stack*. A mesma, funciona como uma pilha, ou seja, quando necessitamos alocar memória, com isso a cada alocação de memória será empilhada. Esse conceito pode ser definido como LIFO (*Last-In-First-Out*).

## II. CONCEITOS FUNDAMENTAIS

Nesta seção, será apresentado os conceitos básicos de pilhas e filas.

### A. Pilhas

Como a própria palavra diz, pilha refere-se a um conceito de pilha mesmo que nós estamos acostumados a ver, pilha de livro como por exemplo, ou seja, algo posicionado em cima de outro. Nos parágrafos seguintes será discutido os conceitos de Empilhar ou *Push* e Desempilhar ou *Pop* do ponto de vista de algoritmos.

A operação de inserção é conhecida como Empilhar ou *Push* que tem como complexidade de  $O(1)$ . O elemento que será inserido, sempre é colocado em cima do último elemento inserido, ou seja, no topo.

A operação de remoção é conhecida como Desempilhar ou *Pop* que tem como complexidade de  $O(1)$ . Nesse caso, somente o primeiro elemento ou elemento do topo ou último elemento inserido será retirado.

Portanto, analisando as operações de inserção e remoção, a pilha segue o conceito de LIFO (*Last-In-First-Out*) como é exemplificado na Figura 1.

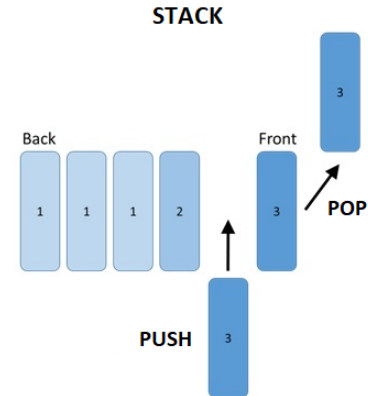


Fig. 1: Representação de pilha com operações de *Push* e *Pop*

### B. Filas

Fila refere-se a um conceito de fila igual ao que nos deparamos no mundo real, fila de um caixa eletrônico como por exemplo. Nos parágrafos seguintes será discutido o conceito de Enfileirar ou *enqueue* e Desenfileirar ou *Dequeue* no cenário de algoritmos.

A operação de inserção é conhecida como Enfileirar ou *Enqueue* que tem como complexidade de  $O(1)$ . O elemento que será inserido, sempre é colocado após o último elemento, ou seja, no final.

A operação de remoção é conhecida como Desenfileirar ou *Dequeue* que tem como complexidade de  $O(1)$ . Nesse caso, somente o primeiro elemento ou elemento da frente será retirado.

Portanto, analisando as operações de inserção e remoção, a fila segue o conceito de FIFO (*First-In-First-Out*) como é exemplificado na Figura 2.

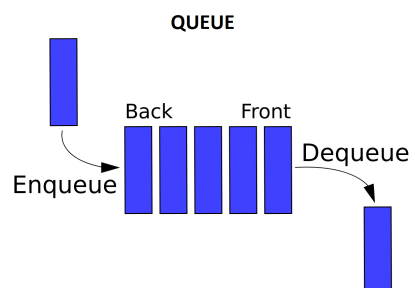


Fig. 2: Representação de fila com operações de *Enqueue* e *Dequeue*

### III. METODOLOGIA

Nesta seção será apresentado a metodologia utilizada para implementar pilha e fila, em subseções diferentes, incluindo pseudocódigos e UML das classes utilizadas implementadas em C++. O código fonte pode ser encontrado em <https://github.com/aparecido/ProgramacaoCientifica>.

#### A. Pilhas

Foi criada uma classe chamada *Stack* que representa a pilha. Essa classe era composta por propriedades e métodos.

As propriedades são *top* (posição do último elemento inserido) e *values* (array de valores da pilha).

Os métodos são *push* (inserir elemento na pilha), *pop* (remover elemento da pilha), *print* (exibir todos elementos da pilha), *printTop* (exibir o último elemento da pilha), *test* (onde foram feito os experimentos que serão apresentados na seção de resultados), *isEmpty* (verifica se a pilha está vazia) e *isFull* (verifica se a pilha está cheia). A figura 3 representa a UML da pilha.

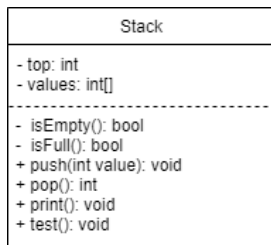


Fig. 3: Representação da UML da Pilha

O algoritmo 1 representa o algoritmo de inserção, *push*.

---

#### Algorithm 1: push - Stack Insertion Algorithm

---

```

if !isFull() then
    values[top] = value;
    top++;
else
    Print "The stack is full.";
end

```

O algoritmo 2 representa o algoritmo de remoção, *pop*.

---

#### Algorithm 2: pop - Stack Deletion Algorithm

---

```

int value = -1;
if !isEmpty() then
    value = values[top - 1]; top--;
else
    Print "The stack is empty.";
end
return value;

```

O algoritmo 3 representa o algoritmo de exibição dos dados existentes da pilha.

---

#### Algorithm 3: print - Stack Show Algorithm

---

```

for i = 0; i < top; i++ do
    Print values[i];
end

```

O algoritmo 4 representa o algoritmo de testes que será discutido na seção de experimentos e resultados.

---

#### Algorithm 4: test - Stack Test Algorithm

---

```

push(1); print();
push(2); print();
push(3); print();
pop(); print();
push(4); print();
push(5); print();
pop(); print();
pop(); print();
pop(); print();
pop(); print();

```

O algoritmo 5 representa o algoritmo de verificação se a pilha está vazia.

---

#### Algorithm 5: stackisEmpty - Stack Is Empty Algorithm

---

```

return top == 0;

```

O algoritmo 6 representa o algoritmo de verificação se a pilha está cheia.

---

#### Algorithm 6: stackisFull - Stack Is Full Algorithm

---

```

return top == SIZE;

```

#### B. Filas

Foi criada uma classe chamada *Queue* que representa a fila. Essa classe era composta por propriedades e métodos.

As propriedades são *first\_position* (posição do próximo elemento a ser retirado), *last\_position* (posição do último elemento da fila) e *values* (array de valores da fila).

Os métodos são *enqueue* (inserir elemento na fila), *dequeue* (remover elemento da fila), *print* (exibir todos elementos da fila), *printNext* (exibir o próximo elemento da fila a ser retirado), *test* (onde foram feito os experimentos que serão apresentados na seção de resultados), *isEmpty* (verifica se a fila está vazia), *isFull* (verifica se a fila está cheia), *addQueue* (adiciona a posição do *last\_position* considerando uma fila circular) e *subQueue* (adiciona a posição do *first\_position* considerando uma fila circular). A figura 4 representa a UML da fila.

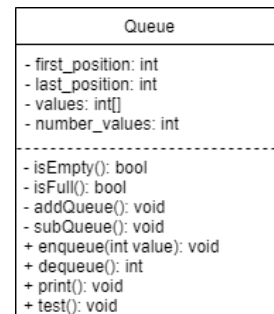


Fig. 4: Representação da UML da Fila

O algoritmo 7 representa o algoritmo de inserção, *enqueue*.

---

**Algorithm 7:** enqueue - Queue Insertion Algorithm

---

```
if !isFull() then
    values[last_position] = value;
    addQueue();
else
    Print "The queue is full.";
end
```

---

O algoritmo 2 representa o algoritmo de remoção, *pop*.

---

**Algorithm 8:** dequeue - Queue Deletion Algorithm

---

```
int value = -1;
if !isEmpty() then
    value = values[first_position];
    subQueue();
else
    Print "The Queue is empty.";
end
return value;
```

---

O algoritmo 9 representa o algoritmo de exibição dos dados existentes da fila.

---

**Algorithm 9:** print - Queue Show Algorithm

---

```
i = first_position;
while i != last_position do
    i = i % SIZE;
    Print values[i];
    i++;
end
```

---

O algoritmo 10 representa o algoritmo de testes que será discutido na seção de experimentos e resultados.

---

**Algorithm 10:** test - Queue Test Algorithm

---

```
enqueue(10); print();
enqueue(9); print();
enqueue(8); print();
dequeue(); print();
dequeue(); print();
dequeue(); print();
dequeue(); print();
```

---

O algoritmo 11 representa o algoritmo de verificação se a fila está vazia.

---

**Algorithm 11:** isEmpty - Queue Is Empty Algorithm

---

```
return number_values == 0;
```

---

O algoritmo 12 representa o algoritmo de verificação se a fila está cheia.

---

**Algorithm 12:** isFull - Queue Is Full Algorithm

---

```
return number_values == SIZE;
```

---

O algoritmo 13 representa o algoritmo para a nova posição do último elemento adicionado.

---

**Algorithm 13:** addQueue - Queue Add Position Algorithm

---

```
last_position = (last_position + 1) % SIZE;
number_values++;
```

---

O algoritmo 14 representa o algoritmo para a nova posição do primeiro elemento a ser

retirado.

---

**Algorithm 14:** subQueue - Queue Sub Position Algorithm

---

```
first_position = (first_position + 1) % SIZE;
number_values--;
```

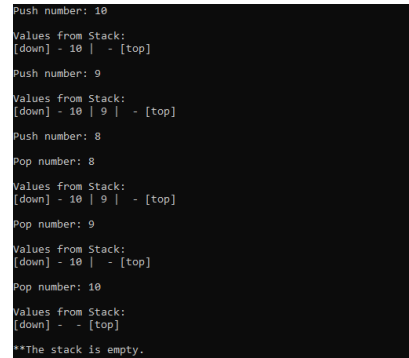
---

## IV. EXPERIMENTOS E RESULTADOS

Nesta Seção será apresentado os experimentos e seus respectivos resultados de forma separadamente.

### A. Pilhas

Foi criado um método chamado *test* que possuía os experimentos, o tamanho da pilha foi definido como 100 posições por meio do *#define*. Foram adicionados 3 números e depois executado 4 remoções, sendo que na última não obteve sucesso pois a pilha já estava vazia. Podemos ver os resultados na Figura 5.



```
Push number: 10
Values from Stack:
[down] - 10 | - [top]
Push number: 9
Values from Stack:
[down] - 10 | 9 | - [top]
Push number: 8
Pop number: 8
Values from Stack:
[down] - 10 | 9 | - [top]
Pop number: 9
Values from Stack:
[down] - 10 | - [top]
Pop number: 10
Values from Stack:
[down] - - [top]
**The stack is empty.
```

Fig. 5: Representação dos resultados da pilha com operações de *Push* e *Pop*

### B. Filas

Foi criado um método chamado *test* que possuía os experimentos, o tamanho da fila foi definido como 100 posições por meio do *#define*. Foram adicionados 3 números e depois executado 4 remoções, sendo que na última não obteve sucesso pois a fila já estava vazia. Podemos ver os resultados na Figura 6.

```

Enqueue number: 10
Values from Queue:
[begin] - 10 | - [end]
Enqueue number: 9
Values from Queue:
[begin] - 10 | 9 | - [end]
Enqueue number: 8
Values from Queue:
[begin] - 10 | 9 | 8 | - [end]
Dequeue number: 10
Values from Queue:
[begin] - 9 | 8 | - [end]
Dequeue number: 9
Values from Queue:
[begin] - 8 | - [end]
Dequeue number: 8
Values from Queue:
[begin] - - [end]
**The queue is empty.
Values from Queue:
[begin] - - [end]

```

Fig. 6: Representação dos resultados da fila com operações de *Enqueue* e *Dequeue*

## V. TRABALHOS RELACIONADOS

Nesta seção será apresentado alguns trabalhos que utilizaram listas ligadas, pilhas ou filas.

Em [5] o autor propõe resolver o problema de ordenar uma sequência de números utilizando uma rede contendo "fila e pilha". Os resultados demonstraram que os autores acharam alguns casos e solucionaram os mesmos, ou seja, não todos e que pela dificuldade encontrada algumas questões não foram respondidas ou solucionadas.

Em [2] o autor propõe implementar a estrutura de dados Pilha e Fila com *Hash*. O autor mostra a aplicabilidade em um exemplo de "Brandes" que é um algoritmo para o cálculo da estatística de centralidade de distância para uma rede social.

Em [3] o autor propõe um novo algoritmo de fila concorrente sem bloqueio e um novo de "fila de dois bloqueios", ou seja, é possível processar *pop* e *push* no mesmo instante. Os experimentos aplicados na "12-node SGI Challenge multiprocessor" demonstraram que o algoritmo proposto supera as soluções conhecidas.

Em [1] o autor propõe uma nova solução para reduzir a perda de pacotes no tráfego de dados na internet comparando com as soluções já apresentadas, em que algumas delas é baseada em Fila. O método apresentado é baseado em fila também, conhecido como *Stochastic Fair BLUE* (SFB).

Em [4] o autor propõe uma nova técnica micro-arquitetônica para reduzir a energia dissipada por filas e pilhas. Os resultados demonstraram que o método pode economizar a energia da fila de instrução em até 30% e a energia da fila de dados de vídeo em 20%.

## VI. CONCLUSÃO

Após a análise do conceito e resultados obtidos de pilha e fila é possível concluir que as duas estruturas possuem a mesma complexidade de ordem  $O(1)$  na inserção e remoção. Além disso, foi mostrado que o algoritmo pode ser utilizado com o objetivo de gerenciar um os dados. Talvez um próximo passo seria a utilização desse mesmo algoritmo para diversos

problemas e comparar o mesmo com outras estruturas, e assim evidenciar os resultados obtidos.

## REFERENCES

- [1] W. chang Feng, K. G. Shin, D. D. Kandlur, and D. Saha. The blue active queue management algorithms. *IEEE/ACM Trans. Netw.*, 10:513–528, 2002.
- [2] L. Hoyle. Implementing stack and queue data structures with sas ® hash objects. 2009.
- [3] M. M. Michael and M. L. Scott. Simple, fast, and practical non-blocking and blocking concurrent queue algorithms. In *PODC*, 1996.
- [4] V. G. Moshnyaga. Energy reduction in queues and stacks by adaptive bitwidth compression. In *ISLPED*, 2001.
- [5] R. E. Tarjan. Sorting using networks of queues and stacks. *J. ACM*, 19:341–346, 1972.