Implementação de Filas e Pilhas em C++ Utilizando Orientação a Objetos

Murillo Freitas Bouzon

***Resumo*—Os conceitos de pilha e fila são utilizados na maioria dos modelos computacionais desde a época em que foram definidos, sendo estruturas de dados de grande importância para a resolução de uma boa parte de problemas relacionados à programação. Com isso em mente, neste trabalho foi proposta a implementação de uma classe de fila e uma classe de pilha, utilizando a linguagem C++ com paradigmas de programação orientada a objetos. As duas classes foram testadas com a inserção e remoção de caracteres, mostrando que foi possível realizar todas as operações corretamente.**

***Index Terms*—Pilha, Fila, C++, Orientação a objetos, Estrutura de dados**

I. INTRODUÇÃO

O uso de estrutura de dados têm sido uma característica recorrente nas diversas aplicações computacionais atualmente. Isso ocorre devido à imensa quantidade de dados que são armazenados e utilizados, tornando necessária uma maneira de organizar e gerenciar esses dados, de acordo com a aplicação em que eles estão sendo utilizados, para facilitar o acesso e a modificação dos mesmos.

Uma estrutura de dados bem conhecida é a pilha, sendo utilizada pela maioria das linguagens de programação para gerenciamento de memória e verificação do estado de sub-rotinas chamadas pelo programa principal, além de ser utilizada em diversas outras aplicações atuais. Outra estrutura de dados também bastante utilizada é a fila, sendo utilizada, por exemplo, em simulação de sistemas ou ambientes do mundo real que possuem filas, CPU *scheduling*, sincronização de dados entre dois processos e na impressão de documentos.

Com o surgimento do paradigma de programação orientado a objetos, a implementação dessas estruturas de dados passou a ser feita dessa forma, permitindo o encapsulamento do código e facilitando o uso de estruturas de dados em diferentes cenários.

Sendo assim, neste trabalho foi feita a implementação de uma pilha e uma fila, utilizando orientação a objetos, com o intuito de entender melhor os conceitos das estruturas de dados implementadas e realizar boas práticas de programação orientada a objeto.

II. FUNDAMENTAÇÃO TEÓRICA

Nesta seção serão apresentadas as teorias relacionadas às estruturas de dados de pilha e fila utilizadas neste trabalho.

A. Pilha

Pilha é uma estrutura de dados do tipo LIFO (*last-in-first-out*), onde o último elemento a ser inserido é o primeiro a ser removido. A sua implementação pode ser feita por meio de um vetor, porém, a manipulação deste vetor é feita em apenas uma de suas extremidades, chamada de topo.

Existem diversas maneiras de implementar uma pilha. no entanto, a utilizada neste trabalho seguiu a teoria de pilha apresentada em [2], possuindo os métodos:

- Empilhar (*Push*): Empilha um elemento na pilha.
- Desempilhar (*Pop*): Desempilha o elemento no topo da pilha.
- Topo (*Top*): Mostra o valor no topo da pilha.
- Vazio (*Empty*): Verifica se a pilha está vazia.

As operações de Empilhar e Desempilhar possuem ambas complexidade $\mathcal{O}(1)$.

B. Fila

Fila é uma estrutura de dados do tipo FIFO (*first-in-first-out*), onde o primeiro elemento a ser inserido é o primeiro a ser removido. Assim como a pilha, também pode ser implementada com um vetor, que é manipulado nas suas duas extremidadesm através de dois índices chamados de *head* e *tail*, sendo *head* o índice do primeiro elemento da fila e *tail* o índice do último.

Ela também pode ser implementada como uma fila circular, onde os índices *head* e *tail* voltam para o começo da fila caso ultrapasse o seu tamanho máximo.

Para a implementação de fila deste trabalho, também foi seguido os método estabelecidos em [2], sendo eles:

- Enfileirar (*Enqueue*): Enfileira um elemento na fila.
- Desenfileira (*Dequeue*): Desenfileira o primeiro elemento da fila.
- Frente (*Front*): Mostra o primeiro elemento da fila (Não está definido em [2]).

As operações de Enfileirar e Desenfileirar possuem complexidade $\mathcal{O}(1)$.

III. METODOLOGIA

Foi feita uma implementação de uma pilha de caracteres, representada pela classe *Stack*, e de uma fila de caracteres, representada pela classe *Queue*. As duas implementações foram feitas na linguagem C++, utilizando paradigmas de orientação a objeto.

A Figura 1 apresenta o diagrama UML da classe *Stack*. Os atributos dessa classe são:

- *top*: elemento no topo da pilha.
- S: vetor de caracteres que armazena os elementos dentro da pilha.

A Figura 2 apresenta o diagrama UML da classe *Queue*. Os atributos dessa classe são:

- *head*: índice do início da fila.
- *tail*: índice do final da fila.
- *length*: número de elementos na fila.
- Q: vetor de caracteres que armazena os elementos contidos na fila.

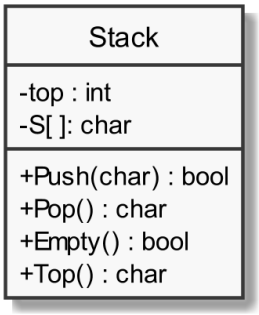

Figura 1: Diagrama UML da classe *Stack*.

O Algoritmo 1 mostra um pseudo-código do método $Empty()$ da classe *Stack*, que retorna *true* caso a pilha esteja vazia e *false* se não estiver.

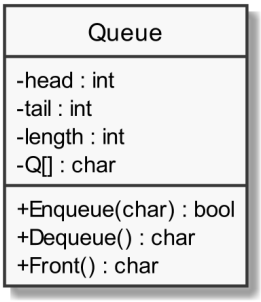

Figura 2: Diagrama UML da classe *Queue*.

Algorithm 1: Stack.Empty()

```
if S.Top() == -1 then
    return TRUE;
else
    return FALSE;
end
```

O Algoritmo 2 mostra um pseudo-código do método $Push(char)$ da classe *Stack*, que recebe um char x e retorna *true* caso a pilha não esteja cheia e foi possível empilhar o elemento x ou *false* se a pilha estiver cheia.

O Algoritmo 3 mostra um pseudo-código do método $Pop()$ da classe *Stack*, que retorna o elemento no topo da pilha ou o valor 0 se a pilha estiver vazia.

O Algoritmo 4 mostra um pseudo-código do método $Enqueue(char)$ da classe *Queue*, que recebe um char x e retorna *true* caso a fila não esteja cheia e foi possível enfileirar o elemento x ou *false* se a fila estiver cheia.

O Algoritmo 5 mostra um pseudo-código do método $Dequeue()$ da classe *Queue*, que retorna o primeiro elemento da fila ou o valor 0 se a fila estiver vazia.

Algorithm 2: Stack.Push(x)

```
if top == SSize - 1 then
    return FALSE;
else
    top++
    S[top] = x
    return TRUE;
end
```

Algorithm 3: Stack.Pop()

```
if Empty() then
    return 0;
else
    top–
    return S[top+1];
end
```

Algorithm 4: Queue.Enqueue(x)

```
if length == QSize then
    return FALSE;
else
    Q[tail] = x
    tail = (tail+1) % QSize
    length++
    return TRUE;
end
```

Algorithm 5: Stack.Pop()

```
if length == 0 then
    return 0;
else
    x = Q[head]
    head = (head+1) % QSize
    length–
    return x;
end
```

IV. EXPERIMENTOS E RESULTADOS

Para validar se as implementações foram realizadas corretamente, foram feitos dois testes, um para cada estrutura de dados implementada.

O primeiro teste foi feito na classe *Queue*, onde definiu o tamanho máximo de 5 e foi enfileirado 5 caracteres e exibidos. Após isso, foi enfileirado mais 1 elemento para verificar se a fila estoura a capacidade máxima. Em seguida, foram desenfileirados todos os elementos e por fim foi removido mais um elemento para verificar se a fila retorna algo após estar vazia. O Algoritmo 6 mostra um pseudo-código do teste descrito.

A Figura 3 mostra o resultado da execução do teste da fila implementada.

O segundo teste foi feito na classe *Stack*, onde foi definido um tamanho máximo de 10 elementos e então empilhou-se 10 caracteres e exibidos. Após isso, foi enfileirado mais 1 ele-

Figura 3: Resultado do teste da classe *Queue*.

Algorithm 6: testeFila(Queue F, char V[])

```
for i = 0 to 5 with step = 1 do
    F.Enqueue(V[i])
end
F.Enqueue(15)
while F.Front() ≠ 0 do
    F.Dequeue()
end
F.Dequeue()
```

mento para verificar se a pilha estoura a capacidade máxima. Em seguida, foram desempilhados todos os elementos e por fim foi removido mais um elemento para verificar se a pilha retorna algo após estar vazia. O Algoritmo 7 mostra um pseudo-código do teste descrito.

Algorithm 7: testePilha(Stack S, char V[])

```
for i = 0 to 10 with step = 1 do
    F.Push(V[i])
end
F.Push(15)
while F.Top() ≠ 0 do
    F.Pop()
end
F.Pop()
```

A Figura 3 mostra o resultado da execução do teste da pilha implementada.

V. TRABALHOS RELACIONADOS

Nesta seção é apresentado alguns trabalhos recentes que utilizaram filas ou pilhas.

No trabalho de [3], foi proposto uma nova abordagem de estrutura de dados que utiliza *timestamp* para evitar ordenações desnecessárias, onde foi implementada uma pilha de alta performance utilizando esta abordagem. O método proposto mostrou que quanto maior a concorrência entre as estruturas de dados, menor é a ordenação, permitindo uma maior performance e escalabilidade.

Em [4], foi proposto um algoritmo para criar uma estrutura de dados do tipo *stack wait-free*. A pilha foi implementada por meio de uma lista encadeada, onde o ponteiro *top* aponta para o topo da pilha. Para cada operação *push*, o elemento é adicionado na lista encadeada e o ponteiro *top* passa a apontar para o novo topo. A operação *pop* não deleta os elementos da pilha, apenas percorre a lista e marca os elementos. Foi mostrado que todas as operações da pilha são *wait-free* e linearizáveis.

No artigo de [1], foi proposto um algoritmo que utiliza uma estrutura de dados de fila circular de baixa complexidade aplicada para segurança de dados. O algoritmo utiliza o tamanho da fila, múltiplas representações de um número de fibonacci e a primeira letra da palavra-chave escolhida. Os resultados mostraram que o algoritmo proposto conseguiu uma complexidade de 50% mais baixo comparado com o método *multiple circular queues algorithm* (MCQA).

VI. CONCLUSÃO

Neste trabalho foi realizado a implementação de duas estruturas de dados conhecidas, fila e pilha, utilizando a linguagem

Figura 4: Resultado do teste da classe *Stack*.

C++ e conceitos de programação orientada a objetos.

Foram realizados dois teste, um para cada estrutura implementada, que mostraram que elas foram implementadas corretamente, sendo possível realizar todas as suas operações corretamente.

Em trabalhos futuros essas estruturas podem ser utilizadas para resolver problemas computacionais, sendo utilizadas em conjunto com grafos, por exemplo.

REFERÊNCIAS

[1] A. N. Albu-rghaif, A. K. Jassim, and A. J. Abboud. A data structure encryption algorithm based on circular queue to enhance data security. *2018 1st International Scientific Conference of Engineering Sciences - 3rd Scientific Conference of Engineering Science (ISCES)*, pages 24–29, 2018.

[2] T. H. Cormen, C. E. Leiserson, R. L. Rivest, and C. Stein. Introduction to algorithms, third edition. 2009.

[3] M. Dodds, A. Haas, and C. M. Kirsch. A scalable, correct time-stamped stack. In *POPL*, 2015.

[4] S. Goel, P. Aggarwal, and S. R. Sarangi. A wait-free stack. In *ICDCIT*, 2016.