Implementação de Filas e Pilhas Dinâmicas Utilizando Lista Ligada Simples e Dupla

Murillo Freitas Bouzon

***Resumo*—Os conceitos de pilha e fila são estruturas de dados de grande importância para a resolução de uma boa parte de problemas relacionados à computação. Uma das formas de implementá-las é utilizando listas ligadas, tornando-as estruturas de dados dinâmicas, podendo utilizá-las sem definir um tamanho prévio. Sendo assim, neste trabalho foi implementado classes de pilhas e filas utilizando listas ligadas simples e duplas, comparando a performance entre elas. Os resultados mostraram que para a inserção, a classe *queueLinked* obteve a pior performance, demorando muito mais que as outras para inserir um novo elemento na fila devido ao uso de lista ligada simples, tendo que percorrer toda a lista para inserir no final da mesma. Para a remoção, as 4 classes testadas obtiveram tempos próximos de 0,025 segundos para remover 100000 elementos.**

***Index Terms*—Pilha, Fila, C++, Lista ligada simples, Lista ligada dupla, 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.

No entanto, quando a implementação dessas estruturas é feita utilizando alocação de memória estática, é necessário definir o número máximo de elementos previamente, correndo o risco de ultrapassar o tamanho máximo durante um empilhamento ou enfileiramento. Para evitar isso, é possível implementá-las utilizando uma abordagem de alocação de memória dinâmica, através do uso de listas ligadas, que aloca um espaço na memória para cada elemento que é inserido na lista, evitando problemas de espaço na inserção.

Sendo assim, neste trabalho foi feita a implementação de uma pilha e uma fila dinâmica, a partir da implementação de listas ligadas simples e duplas, para entender melhor os conceitos de alocação de memória dinâmica e aprimorar as estruturas de dados de pilha e fila feitas anteriormente.

II. FUNDAMENTAÇÃO TEÓRICA

Nesta seção serão apresentadas as teorias relacionadas às estruturas de dados de pilha e fila utilizadas neste trabalho, além dos conceitos de listas ligadas simples e dupla.

A. Lista ligada simples

A lista ligada simples é uma sequência de elementos, chamados de nó, que possui pelo menos dois atributos, o valor e um ponteiro que aponta para o próximo nó da sequência. A lista possui um ponteiro apontando para o primeiro nó da sequência e caso ela esteja vazia o nó aponta para um endereço da memória nulo. A vantagem de utilizar lista ligada simples é porque não é necessário definir previamente o tamanho máximo dela, permitindo que ela cresça e diminua dinamicamente. Para as operações de inserção e remoção, não é necessário deslocar os nós de suas posições, apenas alterar o apontamento entre eles. A Figura 1 mostra um exemplo de uma lista ligada simples que possui 3 nós.

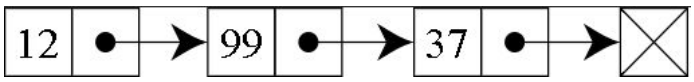

Figura 1: Exemplo de uma lista ligada simples.

Os métodos da lista ligada simples implementadas neste trabalho são:

- *insertBegin*: Insere um novo nó no começo da lista.
- *insertEnd*: Insere um novo nó no final da lista.
- *removeBegin*: Remove o primeiro nó da lista.
- *removeEnd*: Remove o ultimo nó da lista.
- *print*: Imprime todos os elementos da lista.
- *search*: Busca um elemento na lista.
- *isEmpty*: Verifica se a lista está vaiza.

B. Lista ligada dupla

A diferença entre a lista ligada dupla e simples é que os nós da lista dupla possuem apontamento para o próximo nó e para o nó anterior, permitindo uma menor complexidade ao inserir e remover elementos no final da lista. A Figura 2 mostra um exemplo de uma lista ligada dupla que possui 3 nós.

Figura 2: Exemplo de uma lista ligada dupla.

Os métodos da lista ligada dupla são os mesmos da lista ligada simples.

C. 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 $\mathcal{O}(1)$.

D. 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 $\mathcal{O}(1)$ quando implementados com lista dupla. Quando implementado com lista ligada simples, a complexidade para remoção de um elemento é $O(n)$.

III. METODOLOGIA

Neste trabalho foram implementadas 7 classes ao todo. A primeira classe implementada foi a classe Node, possuindo os atributos:

- *value*: O valor do caractere do nó.
- *next*: Ponteiro para o proximo elemento da sequência.
- *prev*: Ponteiro para o elemento anterior da sequência.

Em seguida, foi implementada a classe linkedList, que possui um Node como composição e os seguintes atributos:

- *first*: Primeiro elemento da lista.
- *size*: Número de elementos na lista.

O Algoritmo 1 mostra um pseudo-código do método *insertBegin* e o Algoritmo 2 mostra um pseudo-código do método *insertEnd* da classe *linkedList*.

Algorithm 1: linkedList.insertBegin(char element)

```
Node insertedNode
insertedNode.value = element
insertedNode.next = first
first = insertedNode
```

Algorithm 2: linkedList.insertEnd(char element)

```
Node insertedNode
insertedNode.value = element
Node ptrFront = first, ptrBack = NULL
while ptrFront != NULL do
  | ptrBack = ptrFront ptrFront = ptrFront.next
end
if ptrBack == NULL then
  | insertBegin(element)
else
  | ptrBack.next = insertedNode
  | insertedNode.next = ptrFront
  | size++
end
```

Algorithm 3: linkedList.removeBegin()

```
Node deletedNode = first
first = deletedNode.next
return deletedNode.value
```

Algorithm 4: linkedList.removeEnd()

```
Node ptrFront = first, ptrBack = NULL
while ptrFront.next != NULL do
  | ptrBack = ptrFront ptrFront = ptrFront.next
end
if ptrBack == NULL then
  | ptrBack.next = NULL
  | return ptrFront.value
else
  | removeBegin()
end
```

Os Algoritmos 3 e 4 mostram o pseudo-código dos método *removeBegin* e *removeEnd* da classe *linkedList* respectivamente.

A partir da classe *linkedList*, foram criadas as classes *stackLinked* e *queueLinked*, representando uma pilha dinâmica e uma fila dinâmica que herdam os métodos e atributos de *linkedList*, que possuem os métodos descrito na Seção II-C e II-D.

Além disso, foi implementada a classe *doublyLinkedList*, que representa um lista ligada dupla. Essa classe herda os atributos e métodos de *linkedList* e possui um *Node* por composição. A partir desta classe, foram criadas as classes *stackDoubleLinked* e *queueDoubleLinked* que herdam os atributos e métodos de *doublyLinkedList*.

A Figura 3 apresenta o diagrama UML de todas as classes que foram descritas nessa seção.

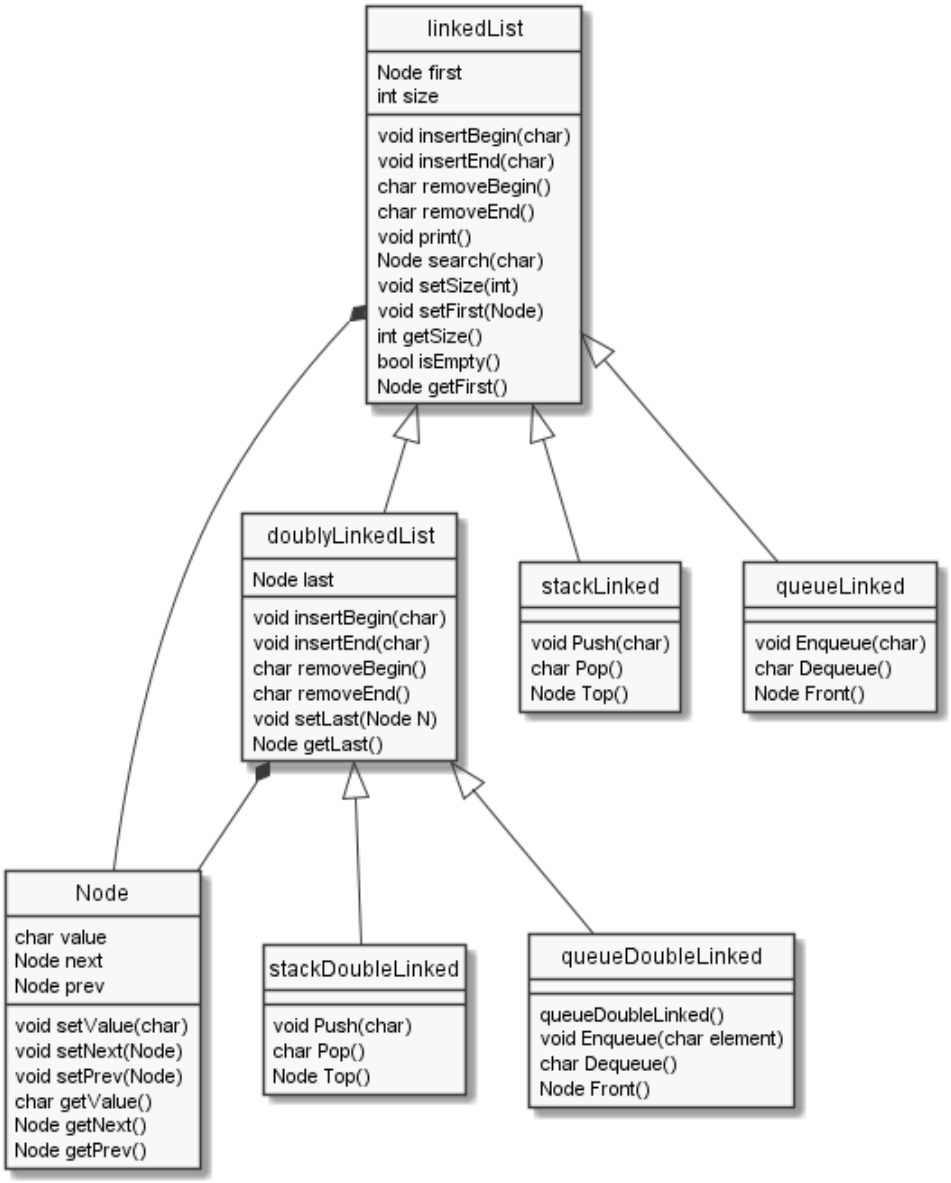

Figura 3: Diagrama UML de todas as classes implementadas.

IV. EXPERIMENTOS E RESULTADOS

Para avaliar as estruturas de dados implementadas, foi feito um experimento para comparar o tempo de inserção e remoção dos elementos com os diferentes tipos de listas ligadas.

Foram inseridos elementos variando de 10 a 100000 e medido o tempo de inserção dos 2 tipos de diferentes de pilhas e filas implementados, totalizando 4 estruturas de dados avaliadas. Foi gerado um gráfico para mostrar os resultados com o tempo de inserção de cada estrutura, onde o eixo x é o número de elementos inserido e o eixo y é o tempo em segundos para inserir todos os elementos. A Figura 4 mostra o gráfico com os resultados do tempo de inserção. Observando esses resultados, pode-se observar que o tempo de inserção da *queueLinked* cresce muito a medida que a quantidade de elementos também aumenta, demorando cerca de 325 segundos para inserir 100000 elementos na fila. Isso ocorre pois é utilizado uma lista ligada simples, ou seja, a *queueLinked* insere sempre no final da lista, precisando percorrer toda a lista para encontrar o último elemento e atualizar o seu ponteiro. Ao trocar por uma lista ligada dupla, a *queueDoubleLinked* consegue inserir com um tempo próximo das outras estruturas de dados implementadas.

Além disso, foi feito um teste também na remoção dos elementos. Foram removidos elementos variando também de 10 a 100000 e medido o tempo de remoção das 4 classes de fila e pilha. A Figura 5 mostra um gráfico gerado com os resultados de tempo de remoção de cada estrutura, onde o eixo x é o número de elementos removidos e o eixo y é o tempo em segundos para remover todos os elementos. Observando o gráfico, pode-se afirmar que as 4 estruturas de dados tiveram tempo muito semelhantes na remoção de elementos, com aproximadamente $0,025$ segundos. No entanto, as estruturas que utilizaram lista ligada simples tiveram um tempo de remoção um pouco menor pois elas realiza uma operação

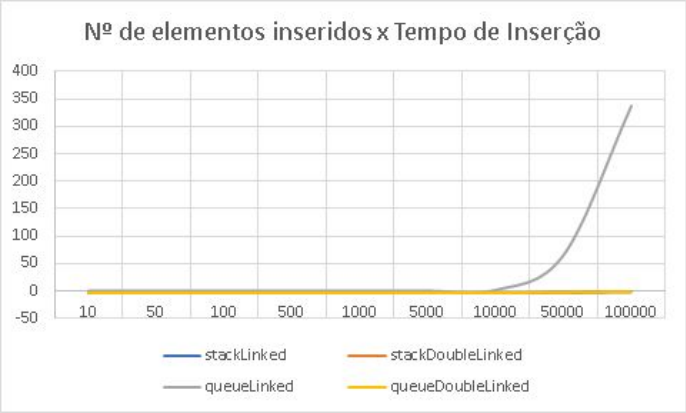

Figura 4: Gráfico com os resultados do tempo de inserção de cada classe.

a menos, já que não precisam atualizar o nó anterior dos elementos.

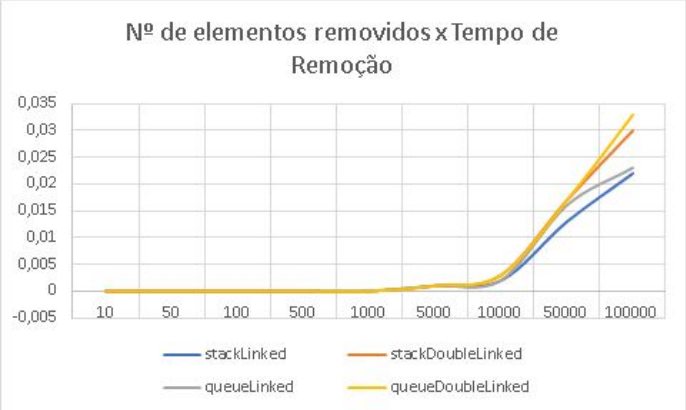

Figura 5: Gráfico com os resultados do tempo de remoção de cada estrutura de dados.

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).

No trabalho de [5], os autores propuseram um jogo instrutivo para auxiliar estudantes a compreender e dominar os conceitos de listas ligadas. O jogo foi desenvolvido em GameMaker Studio. Os resultados mostraram que 93% dos alunos que jogaram tiveram um melhor entendimento de listas ligadas após jogar o jogo.

VI. CONCLUSÃO

Foram implementadas 2 variações de filas e pilhas respectivamente, uma delas utilizando lista ligada simples, sendo chamada de *stackLinked* para pilha e *queueLinked* para fila, e a outra utilizando lista ligada dupla, chamada de *stackDoubleLinked* para pilha e *queueDoubleLinked* para fila.

Para avaliar as classes implementadas, foram realizados dois experimentos. O primeiro mediu o tempo em segundos que as estruturas de dados levam para inserir elementos e o outro mediu o tempo para remover. Os resultados do primeiro experimento mostraram que a classe *queueLinked* possui a pior complexidade na inserção devido ao uso de lista ligada simples para inserção no final da fila, tendo que percorrer toda a fila para encontrar o final dela. O segundo experimentou mostrou que as 4 classes possuem tempos parecidos na remoção, inclusive a *queueLinked*, que remove apenas o começo da fila, por isso obteve um tempo próximo dos outros.

Em trabalhos futuros, essas classes podem ser utilizadas para resolução de problemas de lógica ou até serem utilizadas em outras estruturas de dados mais avançadas. Além disso, é possível utilizar as listas ligadas implementadas aqui para serem estendidas para outras estruturas de dados clássicas.

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.

[5] J. Zhang, M. Atay, E. R. Caldwell, and E. J. Jones. Reinforcing student understanding of linked list operations in a game. *2015 IEEE Frontiers in Education Conference (FIE)*, pages 1–7, 2015.