Pilhas e Filas utilizando Lista Dinâmica Encadeada Simples

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 baseados em Lista Dinâmica Encadeada Simples, 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. Foi possível observar que na remoção e inserção (na pilha, *pop* e *push*, e na fila *queue* e *dequeue*) a complexidade foi diferente pois utilizaram lista dinâmica.**

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 lista dinâmica, pilhas e filas.

A. Lista Dinâmica Encadeada Simples

A lista é uma estrutura de dados que permite você organizar seus dados de forma crescente ou decrescente sendo alocada de forma dinâmica, permitindo escabilidade.

A lista funciona como um encadeamento de nós, onde o nó possui o seu valor ou objeto e um ponteiro para o nó seguinte, permitindo o encadeamento de nós como podemos ver na Figura 1.

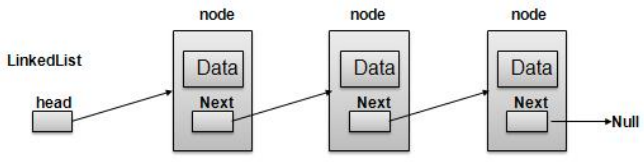

Fig. 1: Representação da lista

A operação de inserção e remoção poder ser feita no em qualquer parte da lista, pois a mesma é ordenada, portanto essa inserção pode acontecer na raíz com apenas 1 elemento na lista, no início, no meio e no fim da lista. Na hora da implementação deverá ser tratado cada cenário de inserção e remoção separadamente. A inserção e remoção tem complexidade de ordem *O(N)* no pior caso e *O(1)* no melhor caso. Exemplificação desses cenários podem ser vistos na Figura 2 operação de inserção e Figura 3 operação de remoção.

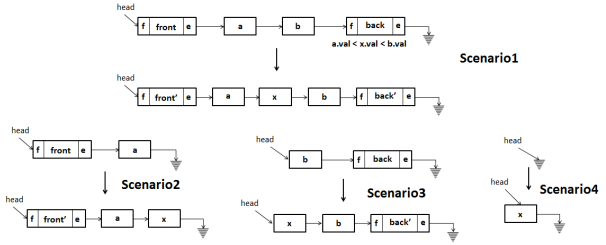

Fig. 2: Representação de cenários de inserção na lista

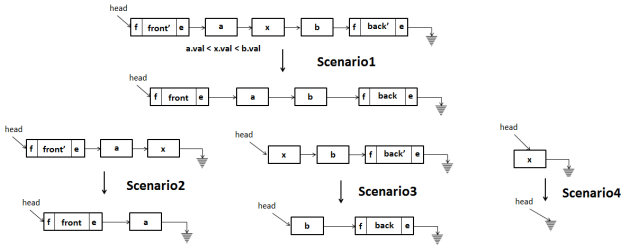

Fig. 3: Representação de cenários de remoção na lista

A operação de busca tem como complexidade de ordem *O(N)* no pior caso e *O(1)* no melhor caso. Consequentemente, a operação de atualização tem mesma complexidade da busca, pois necessita buscar o elemento antes de atualizar.

B. Pilha

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(N)*, se utilizarmos Lista Dinâmica Encadeada Simples. 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)*, se utilizarmos Lista Dinâmica Encadeada Simples. 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 4.

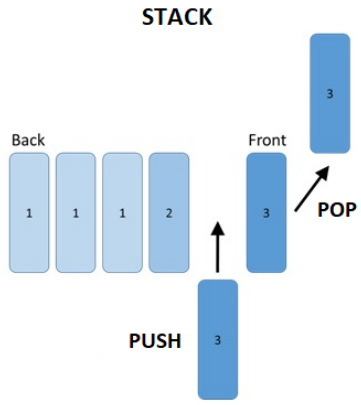

Fig. 4: Representação de uma pilha com operações de *Push* e *Pop*

C. Fila

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(N)*, se utilizarmos Lista Dinâmica Encadeada Simples. 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)*, se utilizarmos Lista Dinâmica Encadeada Simples. Nesse caso, somente o primeiro elemento ou elemento da frente será retirado.

Portanto, analisando as operações de inserção e remoção, a pilha segue o conceito de FIFO *(First-In-First-Out)* como é exemplificado na Figura 5.

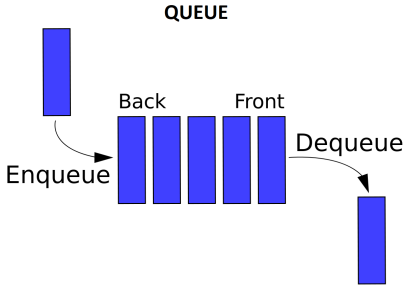

Fig. 5: Representação de uma fila com operações de *Enqueue* e *Dequeue*

III. METODOLOGIA

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

A. Lista Dinâmica Encadeada Simples

Foi criada uma classe chamada *LinkedList* que representa uma Lista Dinâmica Encadeada Simples. Essa classe era composta por propriedades e métodos.

As propriedades são *root_node* (nó que representa a raiz da lista) e *number_nodes* (número de nós existentes na lista).

Os métodos são *addFirst* (adiciona no início da lista), *addLast* (adiciona no fim da lista), *print* (exibir todos elementos da pilha, podendo ser sobreescrito - *override*), *removeFirst* (remove no início da lista), *removeLast* (remove no fim da lista), *getNumberNodes* (retorna o número de nós na lista), *getRoot* (retorna o nó da raíz), *test* (onde foram feitos os experimentos, podendo ser sobreescrito - *override*), *deleteNode* (deleta um nó, limpando a memória), *createNode* (cria um nó alocando memória), *isEmpty* (verifica se a lista está vazia) e *isFull* (verifica se a lista está cheia). A figura 6 representa a UML da lista.

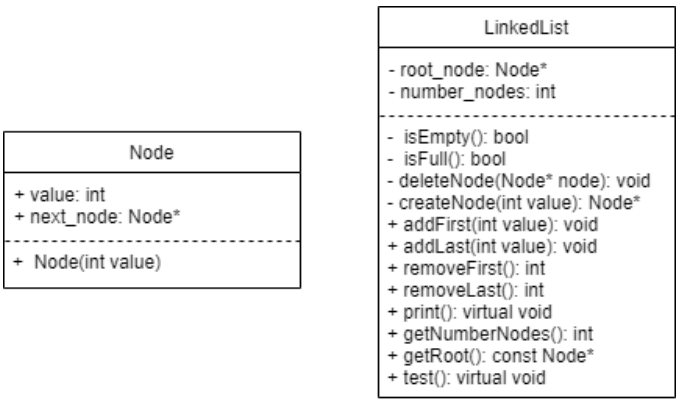

Fig. 6: Representação da UML da Lista Dinâmica Encadeada Simples

O algoritmo 1 representa o algoritmo de inserção no inicio da lista.

O algoritmo 2 representa o algoritmo de inserção no fim da lista.

O algoritmo 3 representa o algoritmo de remoção no começo da lista.

Algorithm 1: addFirst - List Beginning Insertion Algorithm

```
if !isFull() then
    if isEmpty() then
        root_node = createNode(value);
    else
        node = root_node;
        root_node = createNode(value);
        root_node-¿next_node = node;
    end
else
    Print "The list is full.";
end
```

Algorithm 2: addLast - List Ending Insertion Algorithm

```
if isEmpty() then
    root_node = createNode(value);
else
    if !isFull() then
        node = root_node;
        while node-¿next_node != NULL do
            node = node-¿next_node;
        end
        root_node-¿next_node = createNode(value);
    else
        Print "The list is full.";
    end
end
```

Algorithm 3: removeFirst - List Beginning Deletion Algorithm

```
value = -1;
if !isEmpty() then
    node = root_node;
    root_node = node-¿next_node;
    value = node-¿value;
    deleteNode(node);
else
    Print "The list is empty.";
end
return value;
```

O algoritmo 4 representa o algoritmo de remoção no fim da lista.

Algorithm 4: removeLast - Stack Ending Deletion Algorithm

```
value = -1;
if !isEmpty() then
    node = root_node;
    previous_node = NULL;
    while node != NULL do
        if node-¿next_node != NULL then
            previous_node = node;
        end
        node = node-¿next_node;
    end
    if previous_node != NULL then
        previous_node-¿next_node = NULL;
    else
        root_node = NULL;
    end
    value = node-¿value;
    deleteNode(node);
else
    Print "The list is empty.";
end
return value;
```

O algoritmo 5 representa o algoritmo de exibição dos dados existentes da lista.

Algorithm 5: print - List Show Algorithm

```
node = root_node;
while node != NULL do
    Print node;
    node = node-¿next_node;
end
```

O algoritmo 6 retorna quantidade de nós na lista.

Algorithm 6: getNumberNodes - List Number nodes Algorithm

```
return number_nodes;
```

O algoritmo 7 retorna o nó raíz da lista.

Algorithm 7: getRoot - List Root Node Algorithm

```
return root_node;
```

O algoritmo 8 retorna se a lista está vazia.

Algorithm 8: isEmpty - List Empty Algorithm

```
return root_node == NULL;
```

O algoritmo 9 retorna se a lista está cheia.

Algorithm 9: isFull - List Full Algorithm

```
node = createNode(0);
if node == NULL then
    Print "Error: Cannot allocate memory, Node is
      null.";
    return true;
end
this-¿deleteNode(node);
return false;
```

O algoritmo 10 aloca memória do nó.

Algorithm 10: createNode - List Create Node Algorithm

```
return new Node(value);
```

O algoritmo 11 limpa memória do nó.

Algorithm 11: deleteNode - List Delete Node Algorithm

```
if node == NULL then
    delete node;
end
```

B. Pilha

Foi criada uma classe chamada *Stack* que herda de *LinkedList* 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 feitos 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 7 representa a UML da pilha.

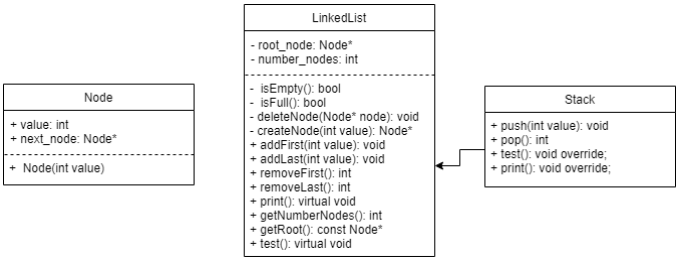

Fig. 7: Representação da UML da Pilha

O algoritmo 12 representa o algoritmo de inserção, *push*.

Algorithm 12: push - Stack Insertion Algorithm

```
LinkedList::addFirst(value);
```

O algoritmo 13 representa o algoritmo de remoção, *pop*.

Algorithm 13: pop - Stack Deletion Algorithm

```
return LinkedList::removeFirst(value);
```

O algoritmo 14 representa o algoritmo de exibição dos dados existentes da pilha.

Algorithm 14: print - Stack Show Algorithm

```
LinkedList::print();
```

O algoritmo 15 representa o algoritmo de testes que será discutido na seção de experimentos e resultados.

Algorithm 15: 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();
```

C. Fila

Foi criada uma classe chamada *Queue* que herda de *LinkedList* que representa a pilha. 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), *test* (onde foram feito os experimentos que serão apresentados na seção de resultados). A figura 8 representa a UML da fila.

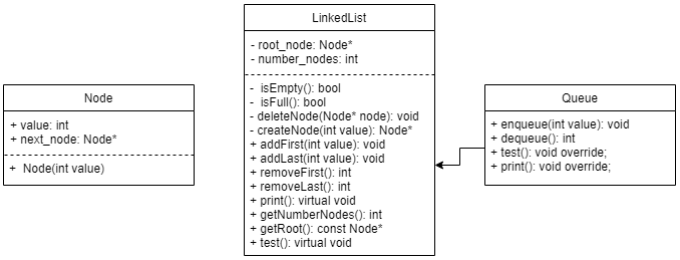

Fig. 8: Representação da UML da Fila

O algoritmo 16 representa o algoritmo de inserção, *enqueue* ou enfileirar.

Algorithm 16: enqueue - Queue Insertion Algorithm

```
LinkedList::addLast(value);
```

O algoritmo 17 representa o algoritmo de remoção, *dequeue* ou desenfileirar.

Algorithm 17: dequeue - Queue Deletion Algorithm

```
return LinkedList::removeFirst(value);
```

O algoritmo 18 representa o algoritmo de exibição dos dados existentes da fila.

Algorithm 18: print - Queue Show Algorithm

```
LinkedList::print();
```

O algoritmo 19 representa o algoritmo de testes que será discutido na seção de experimentos e resultados.

Algorithm 19: Test - Queue Test Algorithm

```
enqueue(1); print();
enqueue(2); print();
enqueue(3); print();
dequeue(); print();
enqueue(4); print();
enqueue(5); print();
dequeue(); print();
dequeue(); print();
dequeue(); print();
dequeue(); print();
```

IV. EXPERIMENTOS E RESULTADOS

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

A. *Pilha*

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 5 números não de forma sequencial e depois aplicado algumas remoções. Podemos ver a sequência de operações no Algoritmo 15 e os resultados na Figura 9.

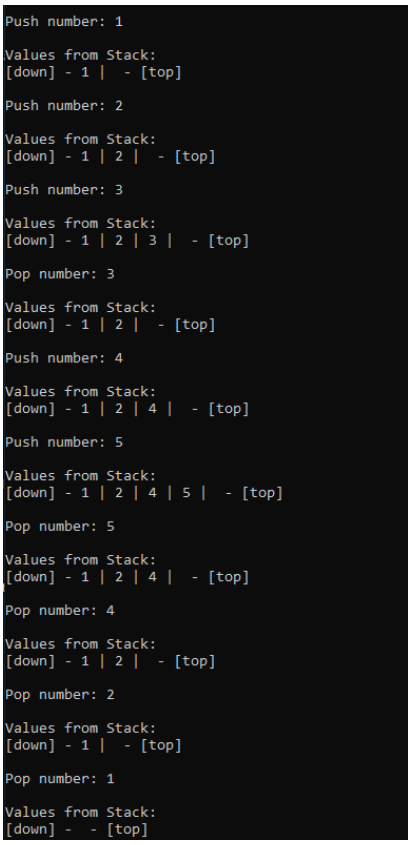

Fig. 9: Representação dos resultados da pilha com operações de *Push* e *Pop*

B. *Fila*

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 5 números não de forma sequencial e depois aplicado algumas remoções. Podemos ver a sequência de operações no Algoritmo 19 e os resultados na Figura 10.

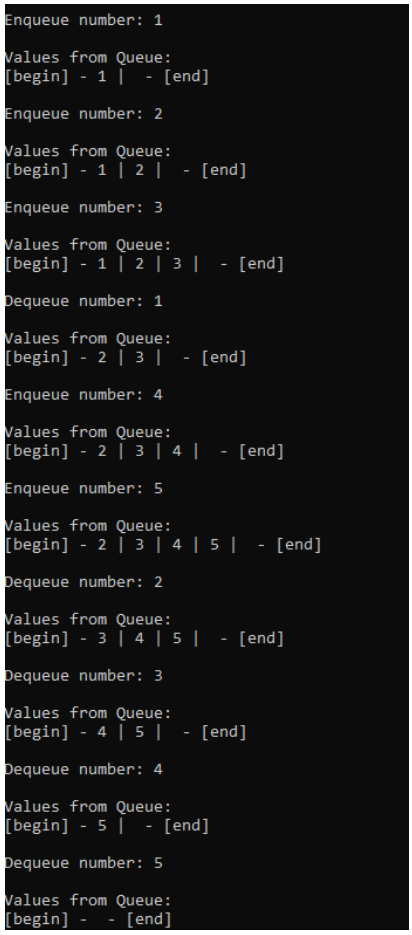

Fig. 10: Representação dos resultados da fila com operações de *Enqueue* e *Dequeue*

V. TRABALHOS RELACIONADOS

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

Em [6] o autor propõe resolver o problema de ordernar 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 perca 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%.

Em [5] o autor faz um estudo baseado na Lista Dinâmica Ligada Simples, ou seja, com apenas um ponteiro para o próximo nó explicando o funcionamento e a implementação da técnica.

Em [7] o autor propõe uma estrutura de dados baseada em lista dinâmica encadeada simples permitindo a concorrência, ou seja, uma lista compartilhada possibilitada de consulta, inserção e exclusão simultâneas.

VI. CONCLUSÃO

Após a análise do conceito e resultados obtidos de pilha e fila é possível concluir que as duas estruturas possuêm complexidade diferentes na inserção e remoção. Além disso, foi mostrado que o algoritmo pode ser utilizado com o objetivo de gerenciar dados. O próximo passo é utilizar Lista Encadeada Dinâmica Dupla para melhorar a complexidade da fila na operação de remoção e talvez 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] P. J. Plauger. A singly linked list. 2000.

[6] R. E. Tarjan. Sorting using networks of queues and stacks. *J. ACM*, 19:341–346, 1972.

[7] J. D. Valois. Lock-free linked lists using compare-and-swap. In *PODC*, 1995.