Trabalho Prático 1

Fernando Eduardo Pinto Moreira - 2019054536

Universidade Federal de Minas Gerais (UFMG)
Belo Horizonte - MG - Brasil

fernandoepm@ufmg.br

1. Introdução

O problema proposto foi implementar um sistema de organização da frota do Imperador Vader para um melhor uso das naves em tempos de guerra. Dado um certo número de naves, deveria-se organizá-las de modo que as naves mais aptas fossem as primeiras a entrar em combate, assumindo que o Imperador sempre informará da nave menos apta para a mais apta. Uma nave em combate pode sofrer processos de avaria, logo, quando isso acontece, a equipe de manutenção envia o código da nave avariada e esta entra na fila para ser consertada, sendo a que foi avariada primeiro será consertada primeiro. Além disso, deve-se oferecer uma lista de naves prontas para o combate e de naves que estão na fila para serem consertadas, tudo isto disponível para consulta a qualquer momento para o Imperador.

2. Implementação

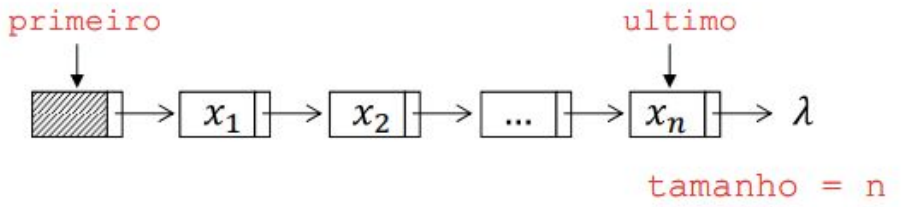
O programa foi desenvolvido na linguagem C++, compilada pelo compilador G++ da GNU Compiler Collection.

2.1. Estrutura de Dados

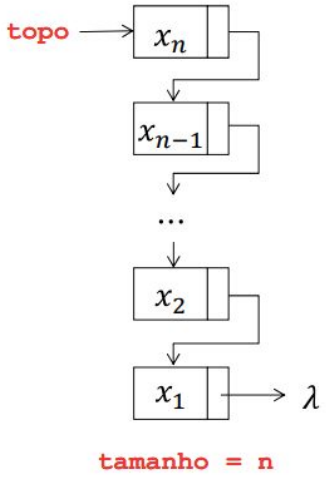
A implementação do programa teve como base as Estruturas de Dados pilha encadeada, lista encadeada e fila encadeada. A escolha de tais estruturas encadeadas se deu devido ao custo de execução do algoritmo, que será constante para inserção e remoção de elementos em tais estruturas (exceto para a remoção em uma dada posição para a lista encadeada, que terá custo linear).

Essas estruturas foram organizadas em classes, cada uma com um nome personalizado para tornar o código mais legível. Neste programa, a classe que implementa a pilha encadeada levou o nome de PilhaNaves, a que implementa a fila encadeada levou o nome de FilaAvaria e a que implementa a lista encadeada levou o nome de ListaCombate. Cada classe herda atributos e métodos da classe pai (pilha encadeada herda de pilha, fila encadeada herda de fila e lista encadeada herda de lista).

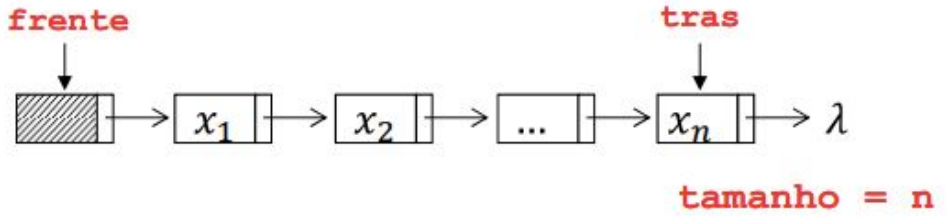
Abaixo, uma ilustração de como uma lista encadeada (ListaCombate) funciona:



Abaixo, uma ilustração de como uma pilha encadeada (PilhaNaves) funciona:



Abaixo, uma ilustração de como uma fila encadeada (FilaAvaria) funciona:



Foi utilizado também uma célula cabeça antes do primeiro elemento em FilaAvaria e em ListaCombate para facilitar e reduzir o custo assintótico de algumas operações.

2.2. Classes

Foram utilizadas oito classes para implementação, dentre elas: TipoItem, TipoCelula, Pilha, PilhaNaves, Lista, ListaCombate, Fila e FilaAvaria. Em TipoItem implementamos um tipo para ser armazenado pelas estruturas. TipoCelula cria uma célula para ser usada pelas estruturas encadeadas. Pilha e PilhaNaves foram usadas para armazenar as naves que estão prontas para entrar em combate. Lista e ListaCombate foram usadas para armazenar as naves que estão em combate e Fila e FilaAvaria foram usadas para armazenar as naves que estão avariadas.

A declaração das classes foi feita em arquivos separados .h e a implementação em arquivos .cpp também separados, isso para aumentar a organização do código e para evitar possíveis erros. Todos esses arquivos estão na pasta headers.

Além disso, foram usados os conceitos de herança em Pilha e PilhaNaves, em Fila e FilaAvaria e em Lista e ListaCombate para aumentar a organização do código, e encapsulamento para evitar que certas variáveis privadas sejam visíveis por todo o programa.

2.3. Funcionamento geral

A ideia geral do código é, a partir de um certo número de naves, colocar as mais aptas em combate, sendo que estas podem sofrer avarias e, quando isso acontece, deve-se consertá-las e a que estiver mais tempo na fila para ser consertada será inserida primeiro em uma lista de naves que estão prontas para o combate.

Isso foi feito, inicialmente, pedindo o número de naves para o usuário. De acordo com este número, lemos os identificadores de todas as naves de forma que as naves menos aptas são lidas primeiro e as mais aptas são lidas por último, obrigatoriamente. Após isso, estas naves devem ser inseridas em uma pilha, pois as mais aptas, ou seja, aquelas inseridas por último, devem ser as primeiras a entrarem em combate. Isso acontece quando o usuário digita o comando 0, onde devemos desempilhar uma nave dessa pilha e inserí-la em uma lista que armazenará todas as naves que estão em combate.

Uma nave em combate pode sofrer avaria e, quando isso acontece, devemos retirá-la da lista de naves em combate e inserí-la em uma fila de naves avariadas, em que a primeira nave a entrar será a primeira a ser consertada, ou seja, será a primeira a sair (por isso foi utilizado a estrutura de dados fila). Uma nave é consertada quando o usuário digita o comando -1 e, quando isso acontece, devemos retirar a nave dessa fila

e inserí-la novamente na pilha de naves que aguardam para entrar em combate. Assim, o processo pode se repetir indefinidas vezes.

2.4. Instruções de compilação e execução

Para compilar o código, acesse o diretório fernando_moreira/src via terminal e digite o comando:

```
make
```

Para isto, você deve ter instalado o make em sua máquina. A instalação varia de cada sistema operacional, mas uma rápida pesquisa na internet resolverá.

Para fazer os testes, acesse o diretório fernando_moreira/src via terminal e digite o comando:

```
make test
```

Para executar o código individualmente (sem a realização da bateria de testes), acesse o diretório fernando_moreira/src via terminal e digite o comando:

```
./tp1
```

Com esse comando, pode-se inserir manualmente uma entrada para o algoritmo.

Para executar o código individualmente baseado num arquivo de entrada, acesse o diretório fernando_moreira/src via terminal e digite o comando:

```
./tp1 < arquivodeentrada.extensao
```

Já para executar o código individualmente baseado num arquivo de entrada e inserir a resposta dada num arquivo de saída, acesse o diretório fernando_moreira/src via terminal e digite o comando:

```
./tp1 < arquivodeentrada.extensao > arquivodesaida.extensao
```

Atenção: o Makefile leva em consideração a sua execução num sistema operacional Linux. Para outros sistemas, como o Windows, adequações podem ser necessárias para a sua utilização.

3. Análise de Complexidade

3.1. Tempo

Começaremos a análise pela leitura do número de naves, logo no início da função main. Percebe-se que tais operações de declaração de variáveis e leitura possuem custo constante, ou seja, O(1).

Logo depois, precisamos ler o número dos identificadores de todas as naves e, para isso, cria-se um for que fará n operações, sendo n o número de naves. Dentro desse for, faz-se a operação de SetChave que possui custo constante e, depois, de empilhar em PilhaNaves, que também possui custo constante. Logo, o custo deste for é:

$$n * O(1) = O(n).$$

Quando o número digitado é zero, temos que desempilhar um elemento de PilhaNaves, operação esta que possui custo constante, inserir este elemento em ListaCombate (neste caso, escolheu-se a função InsereInicio, por ter um custo constante) e após isso, escrever na tela qual nave foi para a batalha, com também custo constante. Logo, quando a operação é igual a zero, temos um custo de:

$$O(1) + O(1) + O(1) = O(1).$$

Quando o número digitado é -1, cria-se um novo TipoItem, com custo O(1), logo após, desenfileira-se um elemento de FilaAvaria (custo constante) e empilha-o em PilhaNaves (custo constante). Depois, escrevemos na tela qual nave foi consertada, com também custo constante. Logo, quando a operação é igual a -1, temos um custo de:

$$O(1) + O(1) + O(1) = O(1).$$

Quando o número digitado é -2, basta chamar a função Imprime criada em PilhaNaves que imprime a pilha de naves prontas para o combate. Esta é uma função recursiva e possui custo linear, pois precisa percorrer todos os elementos da pilha para imprimí-los. Logo, aqui o custo é O(n).

Quando o número digitado é -3, basta chamar a função Imprime criada em FilaAvaria que imprime a fila de naves que estão avariadas. Esta também é uma função recursiva cujo custo é linear, uma vez que percorre toda a fila para imprimir todos os seus elementos. Logo, possui custo O(n).

Já quando o número digitado não é 0, -1, -2 ou -3, induz-se, a partir do enunciado do trabalho, que só poderão ser digitados números correspondentes aos identificadores das naves, cujo objetivo é informar que a nave com tal identificador foi avariada. Primeiramente, escrevemos na tela que a nave com o identificador digitado foi avariada e declaramos uma variável TipoItem auxiliar (ambas operações com custo

O(1)). Depois, para implementar o comando, faz-se uma pesquisa em ListaCombate para pegar o identificador da nave procurada. Aqui, o custo é o mesmo do custo da função Pesquisa implementado em ListaCombate, que é O(1) em seu melhor caso (quando o elemento procurado é o primeiro da lista) e O(n) no pior caso (quando o elemento procurado é o último da lista). O mesmo raciocínio vale para a função PesquisaPosicao que chamaremos um pouco mais abaixo no código, ou seja, que também possui melhor caso O(1) e pior caso O(n). Mas antes de PesquisaPosicao, deve-se enfileirar o elemento pesquisado em FilaAvaria, cujo custo é O(1). Após pesquisar a posição, removemos o elemento de ListaCombate de acordo com a posição encontrada, isto é, chama-se a função RemovePosicao, que possui melhor caso O(1) (quando o elemento a ser removido está na primeira posição) e pior caso O(n), (quando o elemento a ser removido está na última posição). Logo, concluímos que o custo quando o número digitado é X, tal que X é o identificador de uma nave, é dado por:

$$O(1) + O(1) + Máx(O(1),O(n)) + O(1) + Máx(O(1),O(n)) + Máx(O(1),O(n)) = O(n)$$

Logo, conclui-se que o custo de tempo do código é dado por:

$$O(1) + O(n) + O(1) + O(1) + O(n) + O(n) + O(n) = \mathbf{O(n)}$$

3.2. Espaço

Percebe-se que o custo de espaço também será O(n), uma vez que n é o número de "naves"/elementos digitados pelo usuário. Ou seja, apesar de termos 3 estruturas encadeadas, as naves ou estarão em uma, ou em outra ou na terceira, mas nunca em duas ou três delas ao mesmo tempo. Logo, o custo de espaço é igual a: **O(n)**.

4. Conclusão

Pode-se notar, depois da implementação do programa que, desde a escolha das estruturas de dados, até a implementação do programa, tudo é pensado para que o código funcione e compile de forma desejada, mas tão importante quanto isso, que este seja eficiente e que tenha o menor custo possível. Por isso, foram utilizadas estruturas de dados encadeadas como lista, pilha e fila encadeadas, que possuem algumas operações com menor custo.

Além disso, a organização do programa em classes e em arquivos separados, aplicando conceitos de orientação a objetos como herança e encapsulamento, foram fundamentais para que o código ficasse mais organizado, enxuto e coeso. Além disso,

foram feitos comentários onde era mais preciso para quem desejar ler o código e fazer alterações, caso necessário.

Ou seja, pode-se perceber que o trabalho se subdivide em basicamente duas partes, aquela em que criamos as estruturas de dados e aquela em que as usamos. Ambas de fundamental importância para o produto final.

5. Bibliografia

CHAIMOWICZ, Luiz e PRATES, Raquel. Slides da disciplina Estrutura de Dados. Departamento de Ciência da Computação (DCC). Universidade Federal de Minas Gerais (UFMG). 2020.

Ziviani, N. (2006). Projetos de Algoritmos com Implementações em Java e C++: Capítulo 3: Estruturas de Dados Básicas. Editora Cengage.