Trabalho Prático 1
O problema da frota intergaláctica do novo imperador

Euller Saez Lage Silva

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

eullersaez@ufmg.br

1. Introdução

O problema proposto consiste em criar um programa computacional responsável por gerenciar as naves do imperador de um universo paralelo da saga Star Wars durante um combate. Para tal, o imperador deveria ser capaz de poder inserir suas naves das menos até as mais aptas, colocá-las em combate, informar se alguma sofreu avaria e, nesse caso, mandá-la para sua equipe de conserto que daria a manutenção em ordem de chegada, liberando-a para estar disponível para entrar em combate novamente. Além disso, ele também deveria poder visualizar quais naves estão esperando para entrar em combate e quais estão esperando por conserto.

2. Implementação

O programa foi desenvolvido na linguagem C++ por meio do Visual Studio Code para Windows da Microsoft Corporation, utilizando o Windows Subsystem for Linux (WSL) e compilado pelo compilador G++ da GNU Compiler Collection.

2.1 Estrutura de dados

A implementação do programa utilizou as estruturas de dados definidas como lista encadeada, pilha e fila. A escolha por elas se deu justamente por se adequarem melhor a uma resolução mais simples do problema definido, conforme observado nos slides e aulas da disciplina.

No caso das filas, a propriedade FIFO (First In First Out) se mostrou muito útil para armazenar as naves que esperavam conserto.

No caso das pilhas, a propriedade FILO (First In Last Out) se mostrou muito útil para armazenar as naves disponíveis para entrar em combate.

No caso do armazenamento das naves em combate, a lista encadeada se mostrou muito útil por permitir adicionar sempre com custo constante uma nova nave e poder removê-la independentemente da sua posição, o que não ocorre nem nas pilhas e nem nas filas.

2.2 Classes

Foram utilizadas quatro classes principais para a implementação do trabalho.
A primeira delas foi a classe celula que serviu como base para as outras três. Nela declarou-se um atributo _item que armazena um valor inteiro, além de um ponteiro que aponta para outra célula. Como essa é a estrutura base para as demais, todas as outras classes foram declaradas como friends da classe celula, pois assim poderiam acessar seus atributos privados.

As outras classes já citadas são listaEncadeada, pilha e fila.

A listaEncadeada possui dois atributos ponteiros do tipo celula: primeira e ultima. Além disso, possui métodos úteis como um construtor, destrutor, *insere,* que é responsável por receber um inteiro e o inserir numa nova célula no final da lista. Possui também *buscaPosicao,* que recebe um inteiro, busca pela posição da célula que o armazena na lista e retorna sua posição. Por fim, *removeCelula* recebe um inteiro que contém a posição de uma célula na lista e a remove.

A pilha possui um atributo único ponteiro do tipo celula denominado topo.
Seus métodos são apenas um construtor, *empilha*, *desempilha* que retorna um inteiro da celula topo e *imprime*, responsável por imprimir por linha cada item da pilha, do topo até o final.

A fila possui dois atributos ponteiro do tipo celula: primeira e ultima.
Seus métodos são apenas um construtor, *enfilera*, *desenfilera* que retorna o item da primeira celula na fila, além de *imprime*, responsável por imprimir por linha cada item da fila, da primeira celula até a última.

2.3 Operações

Em cada caso de operação entrada pelo imperador, conforme comentado no código, algumas ações deveriam ser tomadas. Em geral, imprimia-se a ação e a armazenava na estrutura correta, como por exemplo quando operação = 0, “nave K em combate” era impresso e K era desempilhada da pilha NavesAptas e inserida na lista NavesEmCombate. Ou quando operação = -1, “nave K consertada” era impresso e K era desenfilerada da fila NavesEmConserto e empilhada na pilha NavesAptas. Nos casos -2 e -3 apenas os métodos imprime da pilha e fila seriam chamados, respectivamente.
Quando operação ≠ 0 ou -1 ou -2 ou -3, isso significaria que a operação continha o id de uma nave, e portanto deveria-se utilizar o método *buscaPosicao* da lista NavesEmCombate, logo em seguida *removeCelula*, “nave K avariada” seria impresso, onde K é sempre o identificador da nave, e K seria enfileirada na fila de NavesEmConserto.

3. Instruções de compilação e execução

3.1 Compilação

Para compilar o programa primeiro é necessário abrir o diretório euller_silva/src no terminal do linux e digitar o comando “make”.

3.2 Execução

Para executar o programa, após compilado, ainda dentro do mesmo diretório euller_silva/src no terminal do linux, basta digitar o comando “./tp1”.

3.3 Limpeza de arquivos temporários

Para limpar todos os arquivos temporários gerados durante a compilação, ainda dentro do mesmo diretório euller_silva/src no terminal do linux, basta digitar o comando “make clean”.

4. Análise de complexidade

4.1 Tempo

Para análise de complexidade de tempo, é possível considerar primeiro os custos que cada método de cada classe têm individualmente para depois analisá-los na função main.

Analisando listaEncadeada, é possível perceber que o método *insere* tem custo constante *O*(1) para qualquer entrada.
O método *buscaPosicao* possui custo constante *O*(1) no seu melhor caso quando a lista possui apenas uma celula ou o item buscado está na primeira célula, e custo linear *O*(n) no seu pior caso que ocorre quando o item buscado não está na lista ou está na última posição, assim como em todos os outros casos que é necessário percorrer toda a lista.
O método *removeCelula* terá custo constante *O*(1) no seu melhor caso quando a celula a ser removida tiver posição = 0 ou custo linear *O*(n) no pior caso que ocorre quando a posição > 0 e o loop do tipo for é executado para percorrer a lista até a posição.

Analisando pilha, é possível perceber que o método *empilha* tem custo constante *O*(1) para qualquer entrada.
O método *desempilha* também possui custo constante *O*(1) para qualquer entrada.
O método *imprime* sempre tem que percorrer toda a pilha, e portanto tem sempre custo linear *O*(n).

Analisando fila, é possível perceber que o método *enfilera* tem custo constante *O*(1) para qualquer entrada.
O método *desenfilera* também possui custo constante *O*(1) para qualquer entrada.

O método *imprime* sempre tem que percorrer toda a fila, e portanto tem sempre custo linear *O*(n).

Assim, a análise da complexidade de tempo do programa como um todo se dá pela soma dos custos de se chamar esses métodos na função main. Portanto, no melhor caso em que apenas entram-se os ids das naves e chama-se apenas as operações 0, -1, ou operação = id de uma nave (considerando que os métodos *buscaPosicao* e *removeCelula* caem em seus melhores casos), o custo é constante *O*(1):

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

No pior caso, teremos todos os custos de chamar os métodos citados no melhor caso (com excessão de *buscaPosicao* e *removeCelula* que agora caem em seus piores casos) mais o custo de se chamar as operações -2 e -3. Assim, no pior caso o custo é linear *O*(n):

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

4.2 Espaço

Vamos considerar que cada celula ocupa uma unidade de espaço no programa. Em todos os métodos de todas as classes o custo de espaço é sempre linear *O*(1). Assim, o custo de espaço na função main será a soma dos custos das chamadas dos métodos das classes, ou seja, *O*(1), mais o custo de se armazenar uma instância de listaEncadeada, uma de pilha e por fim uma de fila. Em todos essas 3 estruturas, o custo de espaço delas será linear *O*(n). Desse modo, o custo de espaço como um todo será linear *O*(n):

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

5. Conclusão

Após o término do trabalho, é perceptível duas fases principais da implementação. A primeira se consistiu em de fato implementar as estruturas de dados escolhidas de forma consistente para a próxima etapa. A segunda fase se definiu em utilizar as estruturas de dados implementadas anteriormente de modo a resolver o problema de gerenciamento das naves do imperador.

Na segunda fase, as principais preocupações eram em quais dessas estruturas armazenar cada nave de acordo com o seu estado, ou seja, se fosse o caso de estar disponível (apta), esperando conserto ou em combate. Além da checagem de qual caso entrado pelo imperador deveria ser executado e como a execução se daria através da manipulação dos métodos implementados nas classes listaEncadeada, pilha e fila.

Portanto, conclui-se que é extremamente vantajoso e produtivo estudar sempre cada caso possível para utilização de diferentes TADs diversos de estruturas de dados. Pois a partir disso abre-se a possibilidade de aproveitar de forma mais precisa e inteligente a modularização disponível para implementação de métodos, por exemplo, que serão muito úteis na resolução de problemas mais específicos.

References

CHAIMOWICZ, L. , PRATES, R. Slides da disciplina Listas Lineares - Estrutura de Dados. Departamento de Ciência da Computação. Universidade Federal de Minas Gerais (UFMG). 2020

CHAIMOWICZ, L. , PRATES, R. Slides da disciplina Pilhas e Filas - Estrutura de Dados. Departamento de Ciência da Computação. Universidade Federal de Minas Gerais (UFMG). 2020

PATIL, S. (2018) “What is a Makefile and how does it work?”, https://opensource.com/article/18/8/what-how-makefile, August.

“How to Create a Simple Makefile - Introduction to Makefiles”. Youtube, uploaded by Paul Programming, 18 May 2015, https://www.youtube.com/watch?v=_r7i5X0rXJk