Trabalho Prático 1
O problema da frota intergaláctica do novo imperador

Jean Lucas Almeida Mota

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

jeanlk@ufmg.br

1. Introdução

O problema a ser enfrentado possui natureza fictícia com base no universo cinematográfico paralelo da saga Star Wars. Neste problema, deve ser implementado um sistema personalizado para o gerenciamento das naves do império galáctico. As naves podem possuir 3 estados: Preparada para batalha; Em combate; Nave Avariada. E o número máximo de naves é de 5000.

O sistema é personalizado pois deve obedecer as seguintes peculiaridades do usuário:

1. Posiciona as naves preparadas para combate em ordem inversa, da menos apta para a mais apta;
2. Ao ser solicitado combate, a nave mais apta deve ter prioridade;
3. A primeira nave avariada em combate tem prioridade de conserto;
4. Uma nave consertada possui prioridade para entrar em combate;

O sistema deve possuir os seguintes comandos: **0** - Envia nave para combate; **X** - onde X é o identificador da nave e indica que a nave X foi avariada em combate; **-1** - Indica que uma nave foi consertada; **-2** - Mostra uma lista de naves preparadas para combate; **-3** - Mostra uma lista de naves esperando conserto.

O programa é finalizado por EOF.

2. Implementação

O programa foi desenvolvido na linguagem C++, compilada pelo compilador G++.

2.1 Estrutura de dados

A implementação do programa teve como base três estruturas de dados: pilhas, listas e filas. Uma para cada estado da nave.

Preparada para batalha usa a estrutura de pilhas arranjo. Essa estrutura arranjo obteve prioridade em relação a encadeada pois o número máximo de naves é fixo, sendo assim desnecessário o uso dinâmico de pilhas.

Combate usa a estrutura listas encadeadas. Por causa do seu custo de inserção no início constante. Neste programa usaremos apenas e inserção no início e a remoção de um item específico da lista. Tendo como base o TAD Lista Encadeada mostrado pelo professor durante as aulas, foi implementado um método a mais, o RemocaoItem(item) que pesquisa um item na lista e o remove.

Nave Avariada usa a estrutura de filas arranjo. A estrutura arranjo foi escolhida pois o número máximo de naves é fixo, sendo assim desnecessário o uso de filas dinâmicas.

2.2 Classes

A fim de modularizar a implementação, 3 TADs foram implementados de forma independente. Além de 4 classes que farão uso desses TADs.

Classe *Preparacao* : Usa o TAD *pilha-arranjo* para organizar as naves em pilha. Possui o método *GetTamanho* que retorna a quantidade de naves inicial, o método *RecebeNave(int nave)* que empilha a nave, o método *RetiraNave()* que desempilha uma nave e a retorna, o método *ImprimeNaves()* que imprime na tela todas as naves preparadas para combate em ordem, dá com mais prioridade para a de menor prioridade. Ainda temos um método *SePrepara()* que não está sendo usado no *main()*, mas serve para facilitar testes manuais, pois nele está implementado comandos de entrada de dados. (Basicamente substitui as entradas de dados do *main()*.

Classe *Combate* : Usa o TAD *lista-encadeada* para receber e organizar as naves que entraram em combate. Possui o método *EnviaNave(int nave)* que recebe uma nave e a insere no início da lista das naves que estão em combate e o método *RecuarNave(int nave)* que recebe o nome de uma nave, faz uma pesquisa na lista

de naves em combate, a retira da lista e depois retorna o nome da nave que foi removida.

Classe *NaveAvariada* : Usa o TAD *fila-arranjo* para receber e organizar as naves que foram avariadas. Possui o método *RecebeNave(int nave)* que recebe uma nave avariada e a enfileira, o método *NaveConsertada()* que desenfileira uma nave da fila de naves avariadas e a retorna como uma nave consertada. E por fim, o método *ImprimeNaves()* que mostra na tela todas as naves avariadas em ordem dá com mais prioridade de conserto para a de menor prioridade.

Classe *SistemaControladorDarkSide:* É a classe de mais alto nível, ela que faz a manipulação das naves usando objetos das classes anteriores. Além de fazer operações dado um comando do usuário. Possui o método construdor *SistemaControladorDarkside(int n)* que apenas atribui a quantidade de naves inicial no sistema, o método *AdicionaNave(int nave)* recebe uma nave e o envia para as naves preparadas para combate. o método *PainelDeComando(int comando)* que recebe um comando e executa a operação desejada, obedecendo às especificações vistas na **1.Introdução**. E ainda temos o método *IniciaAutomatico()* que não está sendo usado pelo *main()*, mas serve para facilitar testes manuais, absorvendo tarefas do *main().*

Observação: As classes dentro da pasta *tads* devem ser usadas apenas pelas classes *Preparacao, combate* e *NaveAvariada*. Essa decisão foi tomada para melhor modularização e coesão do código.

2.3 Operações do usuário

A parte do sistema que faz uso das classes para manipular as naves de acordo com as operações do usuário é um dos trechos mais importantes do código. Uma vez que, uma manipulação errada, pode ocasionar erros lógicos de difícil identificação. O método *PainelDeComando(int comando)* da classe *SistemaControladorDarkSide* faz essa manipulação.

Operação:

- **0 -** Retira uma nave da pilha preparacao através da *preparacao.RetiraNave()* e depois envia essa nave para combate usando *combate.EnviaNave(int nave)*

- **-1 -** Desenfileira uma nave avariada da fila de naves avariadas através da *nave_avariada.NaveConsertada()* e depois a envia para a pilha de naves preparadas para combate usando *preparacao.RecebeNave(int nave)*.
- **-2 -** Imprime na tela a pilha de naves preparadas para combate através da *preparacao.ImprimeNaves()*.
- **-3 -** Imprime na tela a fila de naves avariadas esperando conserto através da *nave_avariada.ImprimeNaves()*.
- **X -** Seja X o identificador de uma nave, retira a nave X da lista de naves em combate através da *combate.RecuarNaves(int nave)* e depois coloca esta nave na fila de naves esperando conserto, usando *nave_avariada.RecebeNave(int nave)*.

3. Análise de Complexidade

3.1 Tempo

Ao inicializar o sistema recebendo as naves preparadas para o combate temos um custo de empilhar *n* itens em uma pilha. Cada item empilhado custa O(1) e temos *n* itens. Dessa forma o custo de inicialização é O(n).

Agora devemos analisar o custo das possíveis operações do usuário. Operação:

- **0 -** Desempilha e insere no início de uma lista encadeada: O(1) + O(1) = O(1).
- **-1 -** Desenfileira e empilha: O(1) + O(1) = O(1).
- **-2 -** Desempilha todos as naves, imprime e depois os empilha de novo:
  - No pior caso temos uma pilha com *n* itens: O(n) + O(n) = O(n).
  - No melhor caso temos uma pilha com 1 item: O(1) + O(1) = O(1).
- **-3 -** Desenfileira todas as naves avariadas, imprime e enfileira novamente:
  - No pior caso temos uma fila com *n* itens: O(n) + O(n) = O(n).
  - No melhor caso temos uma pilha com 1 item: O(1) + O(1) = O(1).
- **X -** Seja X o identificador de uma nave. Pesquisa a nave em uma lista encadeada, remove o item da lista e depois enfileira este item removido em uma fila:
  - No pior caso o item está no final da lista: O(n).
  - No melhor caso o item está no começo da lista: O(1).

3.2 Espaço

A quantidade de naves está limitada de acordo com as especificações do problema (máximo 5000 naves). Dessa forma, como não temos duplicação de

naves, elas apenas são realocadas, o custo de espaço não aumenta. Vamos considerar que cada nave ocupe uma unidade de espaço, para nossa analise. Dessa forma foram usados uma pilha arranjo com tamanho 5000, uma fila arranjo com tamanho 5000 e uma fila encadeada que está limitada ao número de naves disponíveis para alocar (máximo 5000). O(5000) + O(5000) + O(n). Já que o *n* varia, vamos analisar o custo melhor caso e no pior caso.

- Melhor caso: O(5000) + O(5000) + O(1) = O(5000), custo constante.
- Pior caso: O(5000) + O(5000) + O(5000) = O(5000), custo constante.

4. Conclusão

Após a leitura do problema proposto, nota-se que, para desenvolver tal sistema, é necessário uma divisão do problema em duas etapas. A primeira etapa consiste em escolher as estruturas de dados que melhor se adaptam ao problema e implementá-las. Já a segunda etapa consiste no desenvolvimento de classes que fazem bom uso das estruturas.

O grande esforço do problema proposto está, sobretudo, na modularização do código. Uma vez que são necessárias 3 estruturas de dados distintas para 3 estados distintos de um único objeto (nave preparada, nave em combate, nave avariada). Estados estes, que podem ser alterados através das operações do usuário. Dessa forma, refatorações foram feitas a fim de deixar as classes o mais independente possível.

Ao finalizar o sistema proposto, ficou ainda mais evidente a importância das boas práticas de programação (nomes de variáveis, indentação, nomes de funções, comentários). Uma vez que, sem elas, fica muito mais difícil encontrar a solução de um erro lógico.

5. Execução

O programa foi desenvolvido no Windows 10 e testado no Windows 10 e no Linux.

Para execução no Windows foi executado o seguinte comando:
mingw32-make
tp1
Para execução no Linux foi executado o seguinte comando:
make

./tp1
Observação: Os testes deixados pelo professor são executados somente no Linux, através do comando:
make test

References

Chaimowicz, Luiz e Prates, Raquel. Slide sobre Listas, Pilhas e Filas (2020). Disciplina Estrutura de Dados DCC205 UFMG.

Post sobre como fazer arquivo MakeFile. Disponível em <https://www.embarcados.com.br/introducao-ao-makefile/#:~:text=Escrevendo%20as%20primeiras%20linhas%20do%20makefile,-O%20cerne%20do&text=O%20make%20interpreta%20como%20indica%C3%A7%C3%A3o,c%2C%20helloWorld > .Publicado em 11/09/2017. Acessado em 17/09/2020.

Post sobre como detectar o sistema operacional no MakeFile. Disponível em https://stackoverflow.com/questions/4058840/makefile-that-distincts-between-windows-and-unix-like-systems>.Publicado em novembro de 2010. Acessado em 17/09/2020.