O problema da frota intergaláctica do novo imperador
Estrutura de Dados

Rita Rezende Borges de Lima - 2019021760

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

ritarezende@ufmg.br

1. Introdução

O problema proposto nesse trabalho consiste em controlar as naves que compõem a frota intergalática de batalha do imperador. A qualquer momento podemos receber a informação de que uma nava em preparação entrou em combate, uma nave em combate foi avariada, ou que uma nave esperando por reparo foi consertada. Para essa manipulação utilizamos estruturas de dados elementares que representam naves em combate, naves aváriadas e naves prontas para entrar combate.

2. Implementação

O código foi desenvolvido na linguagem *c++* e compilado com *g++ -std=c++11* da GNU Compiler Collection

2.1. Estruturas de Dados Utilizadas

Foram implementadas três estruturas de dados para o trabalho:

- **Pilha**: Armazenamos as naves prontas para entrar em combate na pilha, estas estão ordenadas de acordo com a aptidão para a batalha. No ínicio do programa, a medida que as naves são passadas pelo usuário, estas serão adicionadas a pilha por meio da função *empilha* da biblioteca de pilha. Quando o usuário digitar a operação ”0”, a nave menos apta será retirada da preparação por meio da funçao *desempilha* e adicionada na estrutura seguinte, a lista.

- **Lista**: As naves que entram em combate são adicionadas na estrutura lista. A medida que for passado pro programa o identificador das naves avariadas, a função *remove por chave* retira estas da lista e em seguida elas são adicionadas na última estrutura, a fila.

- **Fila**: As naves que tiverem seu identificador passado pelo usuário são naves danificadas em combate que devem ser adicionadas na estrutura Fila usando a função *enfileira*. Quando o usuário digitar a operação ”-1” a nave adicionada há mais tempo na estrutura será retirada por intermédio da função *desenfileira* e adicionada na pilha, a estrutura de preparação para a batalha.

2.2. Classes implementadas

2.2.1. Lista

A classe lista armazena seu tamanho, e um apontador para seu início e final. Além disso contém os seguintes métodos:

- **O construtor** da classe que apenas iguala a variável tamanho a zero e cria uma nova célula para ser a ”cabeça”.
- **O método *void limpa()*** que com o auxilio de uma célula pivô percorre cada uma das n células da lista e deleta todas.
- **O destrutor** da classe que chama a função *limpa* e deleta a cabeça da lista.
- **O método *void insere final(Nave nave)*** que iguala o nosso ponteiro do final da lista, *ultimo*, com uma nova célula que contém a nave adicionada.
- **O método *void imprime ()*** que percorre nossa lista chamando o método imprime de cada um dos elementos nave, assim imprimindo os identificadores de todas as naves.
- **A função *Nave remove por chave(int identificador)*** que com ajuda de duas células auxiliares percorre a lista procurando uma nave com o mesmo identificador que o passado por parâmetro, remove a nave caso essa seja encontrada e a retorna. Ela também faz com que o ponteiro que apontava para a célula com a nave retirada aponte para a célula que a célula retirada apontava.

2.2.2. Fila

A classe fila armazena seu tamanho, e um apontador para sua frente e trás. Além disso contém os seguintes métodos:

- **O construtor** da classe que apenas iguala a variável tamanho a zero, cria uma nova célula para ser a ”frente” e iguala o ponteiro ”tras” à ela.
- **O método *void limpa()*** que com o auxilio de uma célula pivô percorre cada uma das n células da lista e deleta todas e iguala a variável tamanho a zero.
- **O destrutor** da classe que chama a função *limpa* e deleta a frente da pilha.
- **O método *void enfileira(Nave nave)*** que cria uma nova célula contendo a nave passada como parâmetro, faz a célula que está no final da fila apontar para a nova célula e iguala o ponteiro de ”trás” da fila à nossa nova célula.
- **O método *void imprime ()*** que percorre nossa lista chamando o método imprime de cada um dos elementos nave, assim imprimindo os identificadores de todas as naves.
- **A função *Nave desenfileira()*** que remove a célula da frente da pilha e retorna a nave contida no elemento retirado.

2.2.3. Pilha

A classe pilha armazena seu tamanho, e um apontador para seu topo. Além disso contém os seguintes métodos:

- **O construtor** da classe que apenas iguala a variável tamanho a zero e seu topo à $NULL$.
- **O destrutor** da classe que chama a função *desempilha* até que a pilha esteja vazia.
- **O método *void empilha(Nave nave)***, que iguala o nosso ponteiro de topo à uma nova célula contendo a nave passada como parâmetro.
- **A função *Nave desempilha()*** que remove o topo da pilha e retorna a nave armazenada no antigo topo.
- **O método *void imprime ()*** que percorre nossa pilha chamando o método imprime de cada um dos elementos nave, assim imprimindo os identificadores de todas as naves.

2.2.4. Célula

Existe uma classe célula para cada tipo de estrutura, esta contém um apontador para outra célula e um item de Nave, nossa próxima classe.

2.2.5. Nave

A classe nave não era extritamente necessária, já que só guarda um identificador inteiro que poderia ser guardado na Célula das estruturas. Contudo, com o intuito de fazer o código ficar mais legível, criei essa classe.

2.3. Entrada e Saída do programa

A entrada do programa consiste primeiro de um inteiro que significa a quantidade de naves que irão ser passadas pelo usuário para o programa, seguido dos indentificadores das naves que serão adicionadas pelo usuário. Após essa inicialização o usuário poderá passar diversar operações, sendo estas:

- **0:** a nave do topo da pilha, preparação, irá ser retirada e adicionada na lista, combate.
- **X, tal que x é um identificador de uma nave:** indica que a nave com esse identificador deve ser retirada da lista, combate, e adicionada na fila, naves avariadas.
- **-1:** indica que a nave na frente da fila, naves avariadas, foi consertada e deve ser retirada desta e colocada na pilha, preparação de batalha.
- **-2:** os identificadores de todas as naves na estrutura pilha, preparação para combate, serão impressos na tela.
- **-3:** os identificadores de todas as naves na estrutura fila, naves avariadas, serão impressos na tela.

Ao terminar de passar as operações o usuário deve digitar $CTRL + D$ no terminal para terminar o programa.

3. Análise de Complexidade

Agora que entendemos o funcionamento do programa podemos definir a complexidade de suas funções e de forma geral.

3.1. Tempo

A quantidade de naves, n, e a quantidade de operações, m, influenciam o tempo de execução do programa. Chamaremos a função *empilha* no inicio do programa n vezes e a função *operação solicitada*, m vezes. A complexidade do programa varia de acordo com as operações que são escolhidas pelo usuário, logo convém analisar o custo de cada uma dessas operações:

- **0:** A operação 0 chama as funções *desempilha* e *insere final* ambas com custo constante de modo que a operação é $O(1)$.

- **X, tal que x é um identificador de uma nave:** Esta operação chama as funções *remove por chave* e *enfileira* sendo a primeira com custo linear e a segunda com custo constante de modo que a operação é $O(n)$.

- **-1:** A operação -1 chama as funções *desenfileira* e *empilha* ambas com custo constante de modo que a operação é $O(1)$.

- **-2:** A operação -2 chama a função de impressão da pilha que tem custo linear, de forma que a operação é $O(n)$.

- **-3:** A operação -3 chama a função de impressão da fila que tem custo linear, de forma que a operação é $O(n)$.

Como é possível chamar m operações com n custo o custo total é da forma $n + n * m$ que é o mesmo que $O(n * m)$.

3.2. Espaço

A complexidade espacial irá depender da quantidade de naves, ***n*** , adicionadas pelo usuário. Uma nave só pode estar em uma estrutura por vezes, logo o custo é da forma $O(n)$.

4. Instruções de compilação e execução

Para a compilação do programa basta a utilização do *Makefile* presente dentro da pasta *src* do projeto executando o comando *make* no diretório. Para a execução: *./tp1*.
O algoritmo foi implementado e testado em um ambiente linux Ubuntu:

> Distributor ID: Ubuntu
> Description: Ubuntu 18.04.5 LTS
> Release: 18.04
> Codename: bionic

Utilizando o compilador *g++ -std=c++11*.

5. Conclusões

O desenvolvimento do trabalho pode ser separado em duas partes distintas. A primeira, implementar as estruturas de dados escolhidas de forma correta e coesa e a segunda, utilizar essas estruturas para o sistema pedido pelo trabalho. A partir da implementação da segunda parte foi possível observar formas de como resolver problemas diversos utilizando estruturas como pilha, fila e lista, entender onde a aplicação de cada uma é mais adequada e observar suas vantagens.

6. Bibliografia

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