Trabalho Prático I

O problema da frota intergaláctica do novo imperador

Vítor Archanjo Vasconcelos Chaves
vitorarchanjo@ufmg.br
Matrícula: 2018019877

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

1. Introdução

O problema proposto foi implementar um sistema para organizar as frotas do imperador. Esse sistema é responsável por preparar as naves do imperador para combate, enviá-las para a batalha e para o conserto, se necessário. Durante a preparação para a batalha as naves são adicionadas de maneira crescente de aptidão. Após adicionadas, as naves esperam ordem do imperador para serem enviadas para combate. Em combate, as naves podem sofrer danos e então são mandadas para conserto. Uma vez consertada, a nave volta para junto das naves prontas para o combate, esperando novamente a ordem do 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 e fila, ambas encadeadas. A opção por implementar de forma encadeada deu-se por esta apresentar os mesmos custo entre as principais funções básicas, construtor, inserção e remoção, além de apresentar uma memória extra e tamanho dinâmico.

A **pilha** foi a estrutura de dados escolhida por ter definição FILO (First In Last Out), se encaixando com a necessidade do imperador para a preparação das frotas. A Pilha (stack) foi criada usando uma classe auxiliar *stackCell e* a funcionalidade *template.* A classe *stackCell* atua como uma célula da pilha, onde guarda um valor inteiro e um apontador do próprio tipo, apontando assim para a próxima célula desta pilha.

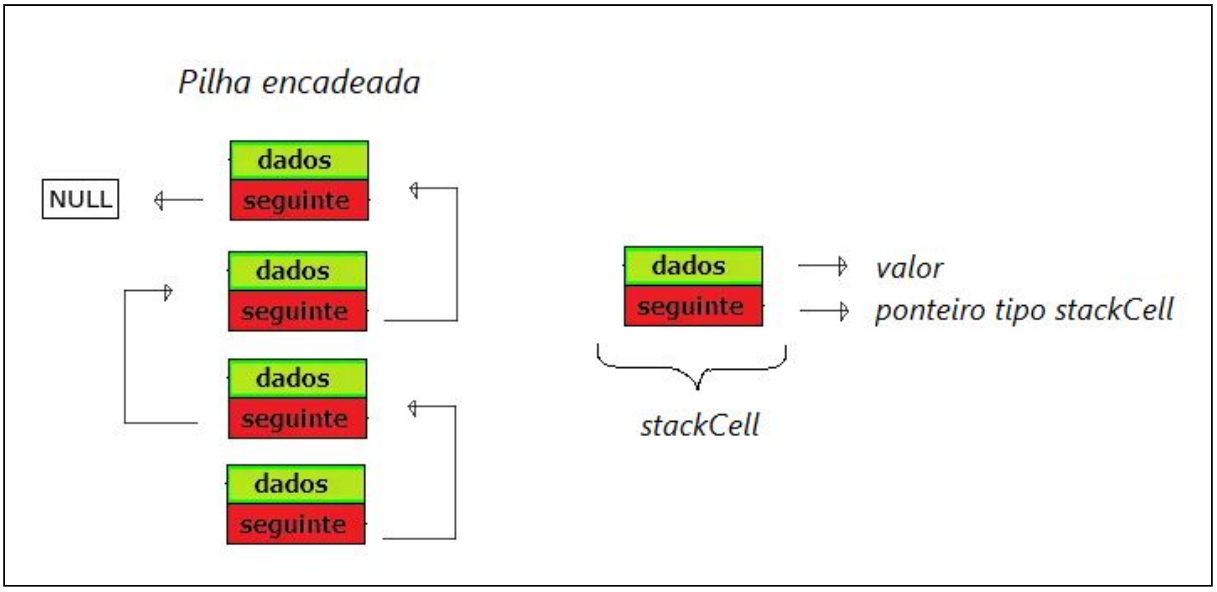

Figura 1. Implementação da Pilha encadeada

O template disponibiliza o uso de uma determinada função/classe, no caso classe *stack*, por um tipo genérico de dados. Na secção 2.2 veremos o porquê e como foi utilizado.

A **lista** foi a estrutura de dados escolhida por ter definição FIFO (First In First Out), sendo utilizada para controlar a frota em combate e as frotas em conserto. A fila (queue) foi criada usando uma classe auxiliar *queueCell e* a funcionalidade *template.* A classe *queueCell* atua como uma célula da fila, onde guarda um valor inteiro e um apontador do próprio tipo. A fila utilizava duas células auxiliares além de uma variável para controlar o valor do tamanho da fila. A primeira célula auxiliar é uma célula *head* (cabeça), definida sempre como a primeira da estrutura. A segunda é a célula *last* (última), que é sempre a última da célula da lista, ou seja, toda vez que é adicionada uma nova célula e last é atribuída com esta célula.

O template disponibiliza o uso de uma determinada função/classe, no caso classe *stack*, por um tipo genérico de dados. Na secção 2.2 veremos o porquê e como foi utilizado.

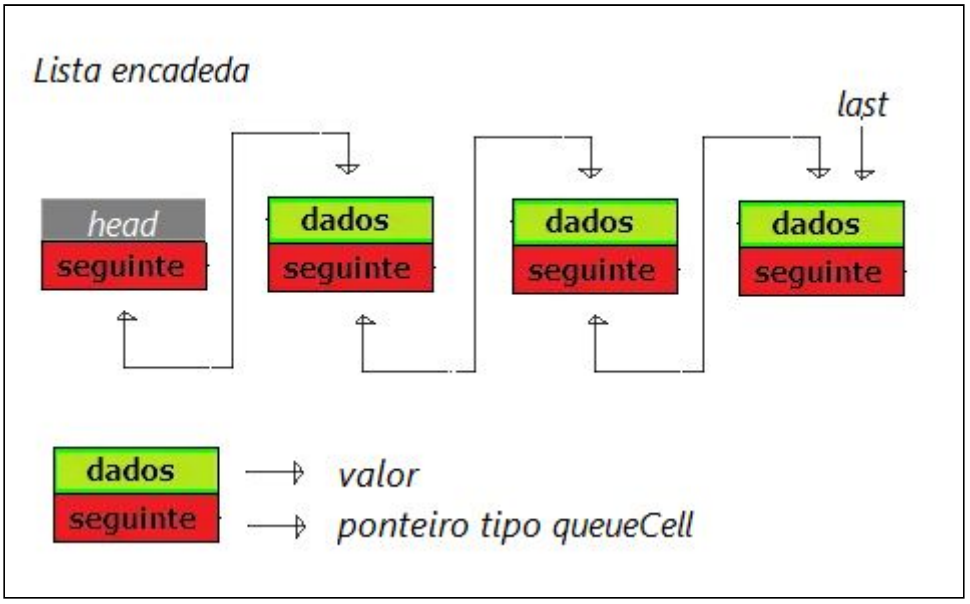

Figura 2. Implementação da Fila encadeada

2.2 Classes

Para modularizar a implementação foram criados três arquivos de cabeçalho (.h), dois para as estruturas de dados abordadas acima (pilha e fila) e outro para a configuração do combate. Para esta configuração foram criadas duas classes: *Spaceship* e *Battle*.

A classe *Spaceship* é criado para fins de organização e compreensão, armazenando apenas o seu id.

A classe Battle é responsável por executar os comandos do imperador e acontecimentos da batalha. Para isso foram criados os métodos: preparingForBattle(), damageShips(), sendForBattle(), repairedShip(), readyForbattle(), inrepair(), que processam o comando do imperador e realizam a função. Além disso, foram criadas duas pilhas e uma fila de tipo Spaceship. Para isso foi usado a funcionalidade template. Dessa maneira foram criadas estruturas que armazenam o tipo de dados Spaceship que armazenava a sua respectiva id.

```
class Spaceship
{
    private:
        int id;

    public:
        Spaceship(int id);
        int getId();
};
```

```
class Battle
{
    private:
        int shipArmy = 0;
        Stack<Spaceship> readyShips;
        Stack<Spaceship> battle;
        Queue<Spaceship> repair;
```

Uso da lista/pilha como template para o tipo de dados genérico criado: Spaceship

Figura 3. Implementação das classes

2.3 Execução dos comandos

A execução do programa se dá de acordo com a entrada de dados (comandos do imperador). As possíveis entradas são: 0, -1, -2, -3, X (Identificador da nave).

Primeiramente há a entrada da quantidades de naves da frota do imperador, e em seguida são adicionadas as naves (via id). Essas naves são adicionadas pelo método preparingForBattle(), que insere as naves na pilha readyShips. Com isso as naves já estão preparadas para o combate.

Para o comando 0 a nave mais apta (do topo da pilha) deve ser enviada para combate. Com isso, o chama-se o método sendForBattle() que remove a nave da pilha readyShips e insere na pilha battle.

Para o comando X (id da nave) a nave com tal id deve ser mandado para conserto, uma vez que foi avariada. Assim o método damageShips() é chamado e então a nave é retirada da pilha *battle* e inserida na fila *repair* (fila de naves a serem consertadas).

O comando -1 indica que a nave a mais tempo adicionada na fila repair foi consertada e portanto deve ser retornada para a fila *readyShips*. Dessa maneira o método repairedShip() é chamado, removendo a nave de *repair* e a inserindo na *readyShips*.

O comando -2 chama o método readyForbattle() que imprime todas as naves em *readyShips* esperando ordem do imperador para entrar em combate.

O comando -3 chama o método inRepair() que imprime todas as naves em *repair* aguardando serem consertadas.

3. Análise de Complexidade

3.1 Tempo

A análise da complexidade do programa pode é feita a partir da execução dos comandos visto na seção 2.3.

Inicialmente é passado o tamanho da frota e em seguida estas são adicionadas à pilha *readyShips.* O comando de inserção das naves na pilha é dado pelo método *stack::insert* que tem como custo constante para a inserção de uma nave. Dado que serão adicionados um número *n* de naves temos:

$$O(1) \times n = O(n)$$

Em seguida o programa executa as funções de acordo com os comandos dados. Estes comandos são:

Adicionar nave ao combate: para adicionar uma nave ao combate o método *stack::remove* é chamado. A remoção de um elemento desta pilha tem custo constante. Além disso é chamado também o método *stack::insert* para adicionar uma nave a batalha, também com custo constante.

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

Naves avariadas: quando avariada, a nave e removida pelo método *stack::remove* que apresenta custo constante e a adiciona na fila pelo método *queue::insert* que também tem custo constante.

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

Naves consertadas: uma vez consertadas as naves são retiradas da fila pelo método *queue::remove* que tem custo constante e adicionadas na pilha *readyShips* pelo método *stack::insert* de custo constante.

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

Impressão das naves prontas para combate: para a impressão de naves prontas para combate o método *stack::print* é chamado. Este por sua vez imprime o id da nave com custo constante. Uma vez que a pilha tem n naves, a função é executada n vezes.

$$O(1) \times O(n) = O(n)$$

Impressão das naves avariadas: para a impressão de naves avariadas o método *queue::print* é chamado. Este por sua vez imprime o id da nave com custo constante. Uma vez que a fila tem n naves, a função é executada n vezes.

$$O(1) \times O(n) = O(n)$$

3.2 Espaço

O programa trabalha com um número fixo de naves, inserido inicialmente pelo usuário. Essas naves são distribuídas pelas estruturas de dados criadas de acordo com os comandos, portanto o custo de espaço do programa é o custo das naves inicialmente inseridas., ou seja $O(n)$.

4. Conclusão

Primeiramente houve uma preocupação com a maneira de implementação da estrutura de dados. Esta devia ser escolhida de forma a facilitar e otimizar os processos necessários na execução do programa. Após a implementação da forma correta das estrutura de dados correta, observou-se a grande aplicabilidade das destas, por sua organização, facilidade de implementação e baixos custo de execução e espaço.

5. Referências

SAVITCH, Walter. Absolute C++. 2015. Sixth edition. Chapter 17: Linked Data Structures. Publishing Company: Pearson.