Trabalho Prático 1

O problema da frota intergaláctica do novo imperador

Gilliard Gabriel Rodrigues
Matrícula: 2019054609

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

gilliard2019@ufmg.br

1. Introdução

O problema proposto foi implementar um sistema de gerenciamento de naves para a frota intergaláctica do imperador Vader. Esse sistema deveria organizar as naves em três estruturas: Preparação para a batalha, Combate e Nave avariada, seguindo as especificações do imperador para cada estrutura. Abaixo temos a especificação de cada estrutura:

• Preparação para a batalha: antes de um combate, as naves são posicionadas pelo imperador de acordo com a sua aptidão para a batalha. Segundo as ordens do imperador, a primeira nave informada ao sistema é a menos apta e, consequentemente, deve ser a última a entrar em batalha. Do mesmo modo, a última nave informada deve ser a primeira a entrar em batalha;

• Combate: dado o comando de Vader para entrar em combate, a nave selecionada deve ser a mais apta que está aguardando, baseado no posicionamento armazenado na estrutura de preparação para a batalha;

• Nave avariada: uma vez que seja reportada que a nave sofreu avaria, ela deve ser excluída da estrutura de naves em combate e ser incluída na estrutura de avaria, onde a nave tentará ser consertada pela equipe de manutenção do imperador. As naves avariadas primeiro são consertadas primeiro. Uma vez que a equipe de manutenção avisa que uma nave foi consertada, a nave passa a aguardar a ordem do imperador para entrar novamente em combate, tendo preferência sobre todas as outras que já estão aguardando.

Com base nessas especificações, será apresentada nesta documentação uma implementação para o sistema baseada nos TAD's pilha, lista e fila vistos em aula. À vista disso, é utilizada uma pilha para estrutura de preparação para a batalha, uma lista para a estrutura de combate e uma fila para a estrutura de nave avariada. Mais detalhes sobre cada implementação serão apresentados no decorrer do trabalho.

2. Implementação

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

2.1 Estrutura de Dados

Como citado anteriormente, a elaboração do programa teve como base as estruturas de dados pilha, lista e fila sequencial. Os três TAD's foram implementados utilizando

arranjos ao invés de apontadores. A preferência pela implementação sequencial ao invés da encadeada deu-se pela sua simplicidade e pelo fato de que o tamanho máximo da frota intergaláctica é conhecido (5000 naves) e, portanto, o tamanho ser dinâmico não é de fundamental importância.

Para a estrutura de preparação, foi utilizada uma implementação de pilha sequencial, pois o comportamento dessa estrutura exigida por Vader se assemelha bastante ao de uma pilha (o último elemento a ser inserido é o primeiro a ser retirado). Além dos métodos de uma pilha vistos em aula, foi implementado um método que imprime os identificadores das naves que estão aguardando para entrar em combate, uma por linha, da nave mais apta até a menos apta, um em cada linha.

Na elaboração da estrutura de combate, foi utilizada uma lista sequencial, uma vez que a possibilidade de retirar um elemento de uma posição qualquer é de suma importância, já que uma nave reportada como "nave avariada" pode estar em qualquer posição, e o TAD lista permite a remoção de elementos do início, do fim e de uma posição informada pelo usuário. Além dos métodos de uma lista vistos em aula, foi implementado um método que percorre a lista em busca de um elemento informado e retorna a posição desse elemento na lista, fundamental para que possamos remover uma nave reportada como avariada de qualquer posição da estrutura.

Por fim, a estrutura de nave avariada foi implementada utilizando uma fila sequencial, já que o comportamento dessa estrutura exigida pelo novo imperador se assemelha bastante ao de uma fila (o primeiro elemento a ser inserido é o primeiro a ser retirado). Além dos métodos de uma fila vistos em aula, foi implementado um método que imprime os elementos da fila do início ao fim, um em cada linha.

2.2 Classes

A fim de modularizar a implementação, foram montadas quatro classes.

A primeira delas chama-se *TipoNave* e foi baseada na classe *TipoItem* apresentada em aula. É possível visualizar seus atributos e métodos na figura 1.

```
class TipoNave{
    public:
        TipoNave();
        TipoNave(int id);
        void SetNaveID(int id);
        int GetNaveID();
        void Imprime();
    private:
        int identificador_da_nave;
};
```

Figura 1. TipoNave.h

As outras três classes são *PilhaDePreparacao*, *ListaDeCombate* e *FilaDeAvaria*, que representam, respectivamente, a pilha, lista e fila sequencial abordadas anteriormente. É possível visualizar seus atributos e métodos nas figuras 2, 3 e 4.

```
class PilhaDePreparacao{
    public:
        PilhaDePreparacao();
        int GetTamanho();
        bool Vazia();
        void Empilha(TipoNave nave_id);
        TipoNave Desempilha();
        void Limpa();
        void ImprimeDoFimAoInicio();
    private:
        int topo;
        static const int MAXTAM = 5000;
        TipoNave estrutura_de_preparacao[MAXTAM];
        int tamanho;
};
```

Figura 2. PilhaDePreparacao.h

```
class ListaDeCombate{
    public:
        ListaDeCombate();
        int GetTamanho();
        bool Vazia();
        TipoNave GetNave(int pos);
        void SetNave(TipoNave nave_id, int pos);
        void InsereInicio(TipoNave nave_id);
        void InsereFinal(TipoNave nave_id);
        void InserePosicao(TipoNave nave_id, int pos);
        TipoNave RemoveInicio();
        TipoNave RemoveFinal();
        TipoNave RemovePosicao(int pos);
        TipoNave Pesquisa(int id);
        int RetornaPosicao(TipoNave nave_id);
        void Imprime();
        void Limpa();
    private:
        static const int MAXTAM = 5000;
        TipoNave estrutura_de_combate[MAXTAM];
        int tamanho;
};
```

Figura 3. ListaDeCombate.h

```
class FilaDeAvaria{
    public:
        FilaDeAvaria();
        int GetTamanho();
        bool Vazia();
        void Enfileira(TipoNave item);
        TipoNave Desenfileira();
        void Limpa();
        void Imprime();
    private:
        int tamanho;
        int frente;
        int tras;
        static const int MAXTAM = 5000;
        TipoNave estrutura_de_avaria[MAXTAM];
};
```

Figura 4. FilaDeAvaria.h

2.3 Entradas e saídas

Conforme especificado pelo novo imperador, a entrada e a saída são padrões do sistema (stdin e stdout, respectivamente). A primeira linha da entrada é composta por um valor inteiro **N** ($0 < \mathbf{N} \leq 5000$), que indica o total de naves da frota do imperador. A seguir, haverão **N** linhas compostas por números inteiros, que representam um identificador de

cada nave. Após, a entrada é composta por várias linhas, cada qual contendo um inteiro I, que indica uma ação a ser realizada no sistema, cujos significados são descritos a seguir:

• 0: indica que o imperador deseja enviar a nave mais apta das que estão aguardando para o combate. Ao ser enviada, a mensagem "nave K em combate" é impressa na saída, na qual K é o identificador da nave;

• X, tal que X é um identificador de uma nave: indica que a nave de identificador X, que estava em combate, foi avariada. Ao ser avariada, a mensagem "nave K avariada" é impressa na saída, na qual K é o identificador da nave;

• -1: indica que a equipe de manutenção informou que uma nave avariada foi consertada. Ao ser consertada, a mensagem "nave K consertada" é impressa na saída, na qual K é o identificador da nave.

• -2: indica que o imperador deseja obter uma impressão do identificador de todas as naves aguardando para entrar em combate, da mais apta para a menos apta. Cada nave é impressa em uma linha;

• -3: indica que o imperador deseja obter uma impressão do identificador de todas as naves avariadas que estão aguardando para serem consertadas. Cada nave é impressa em uma linha, da nave com maior prioridade para ser consertada até a de menor prioridade.

O final da entrada é indicado por *EOF* (Ctrl + D no terminal).

```
#include <cstdio>
#include "TipoNave.h"
#include "PilhaDePreparacao.h"
#include "ListaDeCombate.h"
#include "FilaDeAvaria.h"

int main() {
  PilhaDePreparacao naves em preparacao;
  ListaDeCombate naves_em_combate;
  FilaDeAvaria naves_avariadas;
  TipoNave nave_id;
  int num_naves;
  scanf("%d", &num_naves);
  int i;
  for(i=0; i < num_naves; i++){
    int x;
    scanf("%d", &x);
    nave id.SetNaveID(x);
    naves_em_preparacao.Empilha(nave_id);
  }
  int operacao;
  while(scanf("%d", &operacao) != EOF) {
    switch(operacao){
      case 0:
        nave id = naves em preparacao.Desempilha();
        naves_em_combate.InsereFinal(nave_id);
        printf("nave %d em combate\n", nave_id.GetNaveID());
        break;
      case -1:
        nave id = naves avariadas.Desenfileira();
        naves_em_preparacao.Empilha(nave_id);
        printf("nave %d consertada\n", nave_id.GetNaveID());
        break;
      case -2:
        naves_em_preparacao.ImprimeDoFimAoInicio();
        break;
      case -3:
        naves_avariadas.Imprime();
        break;
      default:
        nave id.SetNaveID(operacao);
        int posicao = naves_em_combate.RetornaPosicao(nave_id);
        nave_id = naves_em_combate.RemovePosicao(posicao+1);
        naves avariadas.Enfileira(nave id);
        printf("nave %d avariada\n", nave_id.GetNaveID());
        break;
    }
  }
  return 0;
```

Figura 5. Arquivo main.cc

Na figura 5 é possível ver a implementação da função main. À medida em que os **N** inteiros são lidos, eles são empilhados na pilha de preparação. O *switch* apresenta 4 *cases* definidos e um *default*.

No *case 0*, uma nave é desempilhada da pilha de preparação e inserida no final da lista de combate, em seguida, o programa imprime na tela a mensagem “nave K em combate”, em que K é o identificador da nave.

No *case –1*, a primeira nave da fila de naves avariadas é removida e empilhada na pilha de preparação, em seguida, a mensagem “nave K consertada” em que K é o identificador da nave, é exibida na saída.

No *case –2*, as naves empilhadas na pilhada de preparação são impressas, da mais apta para a menos apta.

No *case –3*, as naves avariadas aguardando conserto são impressas. Por fim, no default, percebe-se a implementação da ação em que, após o novo imperador inserir o identificador de uma nave, a mesma nave é reportada como avariada, para isso, o método

extra implementado na classe *ListaDeCombate* que pesquisa e retorna a posição de um elemento é chamado, e atribuído a um inteiro *posicao,* em seguida, removemos a nave reportada como avariada da lista de combate (é passado como parâmetro "posicao+1" pois o método que remove uma nave da n-ésima posição recebe sempre a posição como se estivesse começando por 1, e não 0, dentro dela é que esse 1 é decrescido, como na implementação vista em aula), logo após, a nave avariada é inserida no final da fila de naves avariadas e a mensagem "nave K avariada" é exibida, em que K é o identificador da nave.

Conforme solicitado na especificação, é possível ver que a entrada termina no EOF, como é possível visualizar dentro do *while*.

2.4 Instruções de compilação e execução

Para compilar o trabalho, acesse o diretório gilliard_rodrigues/src via terminal e digite:

make

Em seguida, para executar os testes, digite:

make test

Para executar o código manualmente, digite:

./tp1

Logo após, insira o número de naves, em seguida, insira os identificadores das naves. Por fim, como visto na **seção 2.3**, temos as seguintes opções:

Digite *0* para enviar uma nave para o combate;

Digite o identificador de uma nave em combate para reportar nave avariada;

Digite *–1* para reportar que uma nave foi consertada;

Digite *–2* para imprimir os identificadores das naves que estão na estrutura de preparação aguardando para entrar em combate, da mais apta para a menos apta;

Digite *–3* para imprimir os identificadores das naves avariadas que estão aguardando para serem consertadas;

Para finalizar, tecle *Ctrl + D*.

Caso deseje executar o código individualmente baseado em um arquivo de entrada, digite:

./tp1 < arquivodeentrada.extensao

Caso deseje executar o código individualmente baseado em um arquivo de entrada e inserir a resposta dada num arquivo de saída, digite:

./tp1 < arquivodeentrada.extensao > arquivodesaida.extensao

3. Análise de complexidade

3.1 Tempo

Para analisar a complexidade de tempo desse algoritmo, é interessante dar uma olhada na figura 6, que mostra a ordem de complexidade de cada método utilizado na função main.

Figura 6. Análise de Complexidade do main.cc

Através da imagem é possível ver que o custo para ler as **N** linhas com os identificadores de naves é *O(n)*. Embaixo, há um *while* que itera até *EOF* (ou seja, até o usuário teclar Ctrl + D no terminal), dentro desse *while*, existe um *switch* com 4 *cases.* Sabe-se que operações como atribuição a variáveis e exibir uma mensagem na tela têm custo constante, quanto às operações relacionadas aos métodos dos TAD's, é preciso analisar cada caso:

Se o usuário tecla 0, o programa faz uma operação de custo constante, pois desempilhar um elemento de uma pilha sequencial é uma operação de custo *O(1)* e inserir um elemento no final de uma lista sequencial também é *O(1)*;

Se ele tecla –1, também é um custo constante, pois desenfileirar um elemento de uma fila sequencial é uma operação de custo *O(1)* e empilhar um elemento em uma pilha também é *O(1)*;

Se ele tecla –2, o custo é linear, pois para imprimir todos os elementos dá pilha é preciso percorrer ela inteira, logo a operação é *O(n)*;

Se ele tecla -3, o custo também é linear, pois para imprimir todos os elementos de uma fila é preciso percorrer ela inteira, logo a operação é *O(n)*;

No caso em que o usuário digita o identificador de uma nave, é preciso percorrer a lista de combate até encontrar a nave, se a nave for a primeira da lista, a operação é O(1), se for a última, é O(n). Após encontrar a posição e passar como parâmetro para a função que remove a n-ésima elemento, se essa nave estiver na primeira posição da lista, o custo será O(n), pois após removê-la, será preciso deslocar todas as outras naves dentro da lista, caso a nave for a última da lista, o custo será O(1), pois remover o último elemento de uma lista sequencial é uma operação de custo constante. É notável que o melhor caso de uma acontece na mesma situação em que ocorre o pior caso da outra, e vice-versa, logo a ordem de complexidade do *default* do *switch* é *O(n)*, pois

$$O(1) + O(n) = O(n) \text{ e } O(n) + O(1) = O(n).$$

Independentemente do número de vezes em que o usuário escolher fazer as operações, esse número será sempre uma constante, logo, tem-se que o custo de complexidade do while é no máximo $O(n) * k$, que é *O(n)*. Fora do while, há sempre uma operação com custo linear que domina assintoticamente (ler os todos os identificadores de naves), enfim:

$$O(n) + O(n) = O(n).$$

Dessa forma, conclui-se que a ordem de complexidade do algoritmo é *O(n)*, tanto no pior quanto no melhor caso.

3.2 Espaço

Ao instanciar uma pilha sequencial, uma lista sequencial e uma fila sequencial, como o tamanho é fixo, independentemente de o imperador inserir uma nave ou 5000 naves, o custo de complexidade em termos de espaço será o mesmo, ou seja, custo constante ou O(1).

4. Conclusão

O ponto mais importante para conseguir implementar o sistema de gerenciamento de naves para a frota intergaláctica do imperador Vader foi, sem dúvidas, saber qual era a estrutura de dados ideal para cada situação, baseado no comportamento esperado de cada estrutura que o novo imperador especificou, ou seja, identificar que, para a estrutura de preparação de naves, uma pilha deveria ser utilizada, para a estrutura de combate, uma lista deveria ser utilizada e, para a estrutura de nave avariada, uma fila deveria ser utilizada.

Feito isso, o próximo passo era implementar as operações possíveis, como passar uma nave de uma estrutura para a outra em cada situação e imprimir as naves presentes nas estruturas.

No decorrer desse trabalho, ficou evidente a importância de fazer testes para ter certeza de que o seu algoritmo realmente fornece a saída esperada na especificação do projeto, pois se não fossem os testes, alguns erros sutis teriam passados despercebidos, e o fato de o programa estar bem modularizado facilitou a identificação e a correção de bugs.

Referências

Chaimowicz, L., Prates, R. (2020). Estrutura de Dados – TAD's: Listas, Filas, Pilhas e Árvores.