Trabalho Prático I
O problema da frota intergaláctica do novo imperador

Rodrigo José Sousa da Silva

Universidade Federal de Minas Gerais (UFMG)

Belo Horizonte – MG - Brasil

rodrigo7@ufmg.br

1. Introdução

O problema proposto foi implementar um sistema para gerenciamento das naves de batalha do novo imperador Vader. Os objetivo do sistema são: 1) organizar a entrada das naves em batalha de acordo com a sua aptidão, de modo que o imperador possa fornecer uma lista de naves (organizada da menos apta para a mais apta) e elas sejam disponibilizadas para o combate a partir da mesma em ordem inversa e 2) Organizar a fila de reparos para reparar as naves que forem avariadas em batalha, e reinseri-las na fila de naves prontas para o combate.

2. Implementação

O programa foi desenvolvido na linguagem C++(C++11) e compilado utilizando o compilador g++ da GNU Compiler Collection, versão 10.2.0[1]

2.1 Classes

Para implementar o sistema, foram criadas dois tipos de classes, além da classe *main*: a classe ShipManagementSystem, responsável por gerenciar as estruturas de dados e responder aos inputs de operações via *stdin*, e as classes contidas no diretório data/ , que representam as estruturas de dados utilizadas pela classe anterior.

---

1 Para mais informações sobre as configurações da toolchain do compilador, consultar o final do trabalho.

Nessa seção abordaremos apenas a classe **ShipManagementSystem**, as demais serão abordadas na seção seguinte.

A classe ShipManagementSystem foi implementada utilizando o padrão de design Singleton, pois seu intuito é gerenciar e executar as tarefas do sistema de gerenciamento de naves, não permitindo instanciação de mais de um objeto da mesma. Seu construtor instancia as estruturas de dados necessárias (uma lista, uma pilha e uma fila, a serem descritas) e contém as funções para realizar as operações do sistema. Sua instanciação é realizada através da função estática *getInstance()*, que permite verificar se existe uma instância ativa, e caso não acessar o construtor privado para fornecer o endereço para uma. Seus outros métodos são:

- **void operationController(int)** : método que recebe um valor inteiro e realiza a respectiva operação do sistema.
- **void setPreBattleStack(ShipArrayList*)** : método que inicializa a pilha de naves prontas para a batalha.
- **void damageShip(Ship)** : informa que uma nave foi avariada e a insere na fila de reparos.
- **void joinCombat()** : adiciona uma nave da pilha de prontas para o combate para a lista de naves em combate.
- **void repairShip()** : retira uma nave da fila de reparos e a adiciona ao topo da pilha de prontas para o combate.
- **void getShipReserveList()** : lista as naves que estão prontas para o combate.
- **void getRepairBayQueue()** : lista as naves que estão na fila de reparos.

2.2 Estruturas de Dados

Para atender as necessidades do sistema, foram criadas três tipos de estruturas de dados, armazenadas no diretório *data/*, sendo uma delas com duas implementações (uma com ponteiros, outra utilizando um arranjo). As estruturas criadas foram: **ShipArrayList**, **ShipNodeList**, as duas são implementações da interface **ShipList**, e

representam listas; **ShipQueue**, representa uma fila, **ShipStack**, representa uma pilha, e por fim, para as estruturas que utilizam ponteiros, foi criada também a classe **ShipNode**, que representa uma célula dessas estruturas. Além disso, foi implementado um *alias* **Ship** para o tipo *long int* para armazenar o código de identificação das naves.

2.3 Funcionamento do Sistema

Iniciamente a classe *main* prepara as estruturas necessárias para iniciar o sistema, a saber, uma lista (ShipArrayList) para as naves que serão inseridas pelo imperador e uma instância do objeto ShipManagementSystem para gerenciar o resto do processo. Esse objeto também instancia por sua vez três estruturas que serão utilizadas no decorrer do sistema: uma lista (ShipNodeList) para as naves que estão em combate, uma pilha (ShipStack) para as naves que estão prontas para entrar em combate e uma fila (ShipQueue) para as naves avariadas que estão esperando por reparos. Após esse passo, é chamada a função setPreBattleStack(ShipArrayList*), que preenche a pilha de naves prontas para o combate na ordem da mais apta no topo para a baixa no fundo. Após isso, são lidas as operações via *stdin* e a função operationController(int) é a responsável por executá-las. As operações são as descritas na especificação do trabalho: colocar uma nave em combate, enviar uma nave para fila de reparos, tirar uma nave da fila de reparos e inserir no topo da pilha de naves prontas para o combate e por fim, listar as naves prontas para o combate e as naves que estão na fila de reparos.

4. Análise de Complexidade

Para o cálculo de complexidade dos algoritmos, foi analisada a ordem de grandeza de cada método das estruturas utilizadas e da classe de gerenciamento, para as operações de acesso de dados (recuperar um dado na memória), no melhor e pior casos. Para a a análise de espaço, foi considerado quantos espaços cada método/estrutura ocupa utilizando como referência o tipo **Ship**.

Seguem abaixo as tabelas de complexidade por classe. Para a classe *main*, é considerado O(1) para o melhor caso (nenhuma nave, nenhuma operação), e O(k*(2n-1)), onde **k** é o número de operações realizadas (chamadas ao operationController()), e **n** é o número de naves inicialmente inserido no input.

ShipNode	melhor caso	pior caso
setShip	O(1)	O(1)
getShip	O(1)	O(1)
setNext	O(1)	O(1)
getNext	O(1)	O(1)

ShipNodeList	melhor caso	pior caso
getSize	O(1)	O(1)
getByPos	O(1)	O(n)
insertShip	O(1)	O(n)
getShipAndRemove	O(1)	O(n)
clean	O(n)	O(n)

ShipQueue	melhor caso	pior caso
Queue	O(1)	O(1)
getFirst	O(1)	O(1)
clean	O(n)	O(n)
print	O(n)	O(n)

ShipStack	melhor caso	pior caso
stackUp	O(1)	O(1)
pop	O(1)	O(1)
clean	O(n)	O(n)
print	O(n)	O(n)

ShipArrayList	melhor caso	pior caso
getByPos	O(1)	O(1)
getShipAndRemove	O(1)	O(2n-1)
insertShip	O(1)	O(1)
clean	O(1)	O(1)

ShipManagementSystem	melhor caso	pior caso
getInstance	O(1)	O(1)
operationController	O(1)	O(n)
setPreBattleStack	O(n)	O(n)
damageShip	O(1)	O(n)
joinCombat	O(2)	O(2)
repairShip	O(2)	O(2)
getShipReserveList	O(n)	O(n)
getRepairBayQueue	O(n)	O(n)

Para a análise de espaço, podemos considerar as estruturas instanciadas pelo sistema, de acordo com a quantidade de Ships (naves do tipo *long int*) que são requeridas. No total, além dos métodos do programa, são instanciadas quatro estruturas: uma lista utilizando arranjo de tamanho Ship*5000, que armazenará todas as possíveis naves do imperador, uma lista de ponteiros de tamanho **n**, uma pilha de ponteiros de tamanho **m** e uma fila de tamanho **o**. Sendo **p** a quantidade de naves que o imperador disponibiliza, a equação do tamanho ocupado pelo sistema pode ser descrita da seguinte forma:

Se **p = m + n + o**, então o tamanho do sistema será **Ship*(p + 5000)** onde Ship é a quantidade de bytes ocupado por um *long int* em C++.

5. Conclusão

Como conclusão, gostaria apenas de acrescentar um ponto sobre uma decisão de arquitetura que foi importante no projeto do ponto de vista da estrutura de dados. Foi escolhido utilizar duas implementações diferentes de listas para obter um melhor desempenho nos seus casos de uso durante o sistema. Uma implementação utilizando um arranjo foi observada como a melhor decisão para a lista de naves que o imperador disponibiliza para a batalha, porque é uma lista de tamanho máximo fixo. Em outro momento, na lista que representa as naves em batalha, foi escolhido utilizar a implementação com ponteiros, porque é uma lista da qual não sabemos exatamente o tamanho, mas que nunca chegará tão perto do valor máximo da lista anterior.

6. Referências

Cormen , T., Leiserson, C, Rivest R., Stein, C. Introduction to Algorithms, Third Edition, MIT Press, 2009. Versão Traduzida:Algoritmos – Teoria e Prática3a. Edição, Elsevier, 2012.

A toolchain utilizada pelo compilador para compilar o programa relativo ao trabalho foi a descrita abaixo:

```
COLLECT_GCC=g++
COLLECT_LTO_WRAPPER=/usr/lib/gcc/x86_64-pc-linux-gnu/10.2.0
/lto-wrapper

Target: x86_64-pc-linux-gnu
Configured with: /build/gcc/src/gcc/configure --prefix=/usr
--libdir=/usr/lib                    --libexecdir=/usr/lib
--mandir=/usr/share/man  --infodir=/usr/share/info  --with-
bugurl=https://bugs.archlinux.org/  --enable-languages=c,c+
+,ada,fortran,go,lto,objc,obj-c++,d   --with-isl   --with-
linker-hash-style=gnu     --with-system-zlib     --enable-
__cxa_atexit --enable-cet=auto --enable-checking=release --
enable-clocale=gnu  --enable-default-pie  --enable-default-
ssp   --enable-gnu-indirect-function   --enable-gnu-unique-
object --enable-install-libiberty --enable-linker-build-id
--enable-lto  --enable-multilib  --enable-plugin  --enable-
shared --enable-threads=posix --disable-libssp --disable-
libstdcxx-pch   --disable-libunwind-exceptions   --disable-
werror gdc_include_dir=/usr/include/dlang/gdc
```