Sistema de Suporte de uma Rede Ferroviária

Algoritmos e Estruturas de Dados

Ano letivo 2024/2025

Relatório do Trabalho Prático

Leandro Rodrigues 68211

Rodrigo Sinde 68614

# **1 Introdução**

# Com este trabalho pretendemos abordar as várias ED’s introduzidas nas aulas tal como as complexidades temporais das mesmas e a complexidade espacial do programa. Este trabalho também serviu para podermos saber como melhorar um determinado programa em questões de desempenho.

# 

# **2 Descrição**

Este projeto simula o funcionamento de uma rede ferroviária, permitindo a gestão de linhas, estações e horários de comboios. As linhas são compostas por uma sequência de estações, e os utilizadores podem adicionar ou remover linhas e estações, bem como consultar quais estações pertencem a uma linha e quais linhas passam por uma estação. Além disso, o programa gere os horários dos comboios, permitindo adicionar e remover horários associados a linhas específicas, listar horários disponíveis e determinar o melhor horário para um percurso baseado numa hora de chegada pretendida. O sistema suporta a persistência de dados para garantir que as informações são mantidas entre sessões e oferece comandos para gestão e consulta de todos os elementos da rede ferroviária.

# **3 Classes**

Usamos 6 classes para realizar o projeto:

* RailNetworkClass
* LineClass
* StationClass
* StationComparator
* ScheduleClass
* TimeClass

### **3.1 TAD RailNetwork**

O TAD RailNetwork é uma interface que representa a estrutura e o funcionamento de um sistema ferroviário e tem como função receber informações provenientes da classe main e das restantes classes. É na classe RailNetworkClass que são adicionadas as linhas, estações e horários, bem como a remoção e consulta dos mesmos. Também fornece à Main as informações necessárias para imprimir os outputs pedidos pelo utilizador. Usamos para guardar as linhas(lines) e as estações(stations) a estrutura de dados, InsensitiveSepChainHashTable(que implementa a TAD Dictionary), visto que não precisamos de manter uma ordem dos mesmo e sendo mais fácil de inserir, remover e encontrar elementos do que na TAD List.

A InsensitiveSepChainHashTable é feita com base na SepChainHashTable sendo que a Insensitive é feita para não haver distinção entre letras maiúsculas e minúsculas.

**3.2 TADs Line e LineGets**

Estas duas TADs são fundamentais para modelar linhas no sistema ferroviário, havendo uma separação entre as operações de consulta (LineGets) e de modificações(Line).

A classe LineClass serve para armazenar e tratar as informações de uma linha, tal como o seu nome, as schedules e as stations associadas a essa linha. Para isto usamos as estruturas de dados DoubleList e 2 AVLTree para as stations e schedules(departure and arrival), respetivamente.

Usamos para guardar as estações(stations) uma DoubleList<Station> pois aqui é necessário manter a ordem de inserção das estações, pois precisaremos de iterar num método.

Usamos para guardar todos os horários de uma linha duas AVLTree’s uma vez que é necessário ter os horários ordenados. Esta ordenação é feita pela de hora de partida, e temos uma árvore para um sentido e outro para o contrário.

**3.3 TADs Station e StationGets**

Estas duas TADs são fundamentais para modelar estações no sistema ferroviário, havendo uma separação entre as operações de consulta (StationGets) e de modificações(Station).

A classe StationClass tem como função tratar as informações de uma estação, como o seu nome, as linhas a qual esta está associada e os comboios que passam pela mesma. Para isto usamos as estruturas de dados AVLTree e AVLWithComparator para as linhas e comboios respetivamente, que implementam a TAD OrderedDictionary.

Usamos para guardar as linhas da estação uma AVLTree<Line,Line> pois queremos listar em ordem lexicográfica todas as linhas de uma estação.

Usamos para guardar o número dos comboios e o tempo a que passa nessa estação uma AVLWithComparator<Integer,Time> onde este faz a ordenação pelo valor(Time) em vez da chave(Integer) só em caso de empate ele compara pela key.

Esta estrutura de dados é uma extensão da AVLTree, onde foi reescrito os métodos que eram usados os compareTo e substituídos por compare (metodo da classe StationComparator).

**3.4 TAD Comparator**

Esta TAD é responsável por fazer a comparação entre 2 elementos para a ordenação da AVLWithComparator.

A classe StationComparator serve para uma ordenação correta da AVLWithComparator, em que o método na classe implementado compara os valores(Time), e em caso de serem iguais compara as chaves(Integer).

**3.5 TADs Schedule e ScheduleGets**

Estas duas TADs são fundamentais para modelar horários no sistema ferroviário, havendo uma separação entre as operações de consulta (ScheduleGets) e de modificações(Schedule).

A classe Schedule atualiza e organiza os dados essenciais de um horário para um específico comboio. Para isto usamos a estrutura de dados DoubleList de Entrys para os <Station,Time> e uma inteiro como identificador do Comboio.

Usamos para guardar o horário de um comboio uma DoubleList<Entry<Station,Time>> para manter a ordem de inserção das estações e das datas num horário.

**3.6 TAD Time**

A TAD é fundamental para a manipulação de horários no sistema ferroviário.

Esta classe serve para acessar e comparar horas e minutos para que as restantes classes a possam usar nos seus métodos.

# **4 Comandos**

Ao inicializar o programa, é chamado o método load() da classe Main que cria o sistema com as informações guardadas do programa anterior que estão no ficheiro stored.dat.

* **insertLine** - Inserção de linha
* **removeLine** - Remoção de linha
* **consultStations** - Consulta das estações de uma linha
* **consultLine-** Consulta das linhas de uma estação
* **insertTime-** Inserção de horário
* **removeTime -** Remoção de horário
* **consultTimeOfLine-** Consulta dos horários de uma linha
* **consultSchedulesByStation-** Comboios por estação
* **bestTime -** Melhor Horário
* **quit –** Termina a execução do programa

**4.1 Inserção de Linha**

Recebemos o nome da linha e enviamos para a railNetwork que verifica se a linha já existe. Se já existir, lança uma exceção senão -> ele cria uma linha nova, e com a queue de String(que é o nome das estações da linha) dada vamos ver se a estação já existe. Se não existir ele cria uma nova estação e insere à HashTable<String,Station>(que está na railNetwork) e depois adicionamos a estação à linha, a linha à estação, e a linha à HashTable<String,Line>(que está na railNetwork).

**4.2 Remoção de Linha**

Recebemos o nome da linha que queremos remover e enviamos para a railNetwork. Verifica-se se a linha existe na HashTable de linhas e se não existir lança se uma exceção. Se a linha existir chamamos um iterator de todas a estações de uma linha, depois chamamos um método que remove os horários da linha das estações que os contêm.Com o iterador que chamamos vamos verificar as estações da linha, vemos se ela só tiver uma linha significa que é a linha que vamos remover assim removemos essa estação da HashTable de estações da railNetwork, senão removemos a linha da estação.

**4.3 Consulta das estações de uma linha**

Recebemos o nome da linha que queremos consultar e enviamos para a railNetwork. Verifica-se se a linha existe na HashTable de linhas e se não existir lança se uma exceção. Se a linha existir chamamos o método getStationsIterator dessa linha que devolve-nos o iterator de estações.

**4.4 Consulta das linhas de uma estação**

Recebemos o nome da estação que queremos consultar e enviamos para a railNetwork. Verifica-se se a estação existe na HashTable de estação e se não existir lança se uma exceção. Se a estação existir chamamos o método getLinesIterator dessa estação que devolve-nos o iterator de linhas.

**4.5 Inserção de de horário**

Recebemos o nome da linha, o nome do comboio e uma queue de Entry<String,String>

com todos os nomes da estação e o tempo como String e enviamos para a railNetwork. Verifica-se se a linha existe na HashTable de linhas e se não existir lança se uma exceção. Se existir chamamos o método addSchedule da linha. Este método verifica se é uma estação terminal ou não, e se não for lança uma exceção(depois verificamos para qual sentido vamos ter de adicionar o horário). Se for terminal cria-se uma DoubleList de Entry<Station,Time>. Ao iterar vai fazendo verificações para adicionar à lista(ou não) ou lançar a exceção. No final verifica mais algumas verificações e lança exceções dependendo das verificações ou adiciona à AVL o horario de partida e a lista de Entrys e a todas as estações o seu comboio e hora correspondente.

**4.6 Remoção de Horário**

Recebemos o nome da linha, o nome da estação de partida e a hora de partida e enviamos para a railNetwork. Verifica-se se a linha existe na HashTable de linhas e se não existir lança se uma exceção. Se existir, vamos buscar a estação de partida à HashTable da railNetwork e a linha chama o método removeSchedules que recebe a estação de partida e o tempo. Esse método vai ter que verificar qual dos sentidos é a da estação dada e depois faz a remoção à AVL adequada.

**4.7 Consulta de horários de uma linha**

Recebemos o nome da linha, o nome da estação de partida e enviamos para a railNetwork(getSchedulesOfStartStation). Verifica-se se a linha existe na HashTable de linhas e se não existir lança se uma exceção. Se a linha existir, chamamos o metodo getIteratorOfStartStation da LineClass e mandamos a estação de partida(que fomos buscar à HashTable de estações na railNetwork). Verificamos se a estação dada é uma terminal ou não,se não for lança-se uma exceção.

**4.8 Comboios por estação**

Recebemos o nome da estação e enviamos para a railNetwork(getSchedulesByStation). Verifica-se se a estação existe na HashTable de estação e se não existir lança se uma exceção. Se existir essa estação devolve o iterador dos comboios que passam naquela estação.

**4.9 Melhor Horário**

Recebemos o nome da linha, o nome de uma estação inicial e uma final e o tempo de expectável, depois enviamos para a railNetwork(getBestTime). Verifica-se se a linha existe na HashTable de linhas e se não existir lança se uma exceção. Vamos buscar também na HashTable as estações que nos foram dadas, para enviar, juntamente com o tempo, para o método getBestSchedule chamado pela linha. Este metodo vai verificar se ambas as estações existem na linha ou se existem de facto, lançando as exceções adequadas a cada uma. Depois cria um iterador da AVL departureSchedules, e cada horário chama um método que retorna o inteiro, e consoante esse retorno ele retorna o melhor horário ou lança uma exceção. Se ele não encontrar em na departureSchedules vai iterar na outra AVL.

**4.10 Terminar aplicação**

Terminamos a aplicação guardando o estado da mesma através do método save antes de a finalizar.

**5 Complexidades Temporais e Espaciais das funcionalidades**

**5.1 Inserção de Linha**

Melhor caso: O(1)

Pior caso: O(n) sendo n o número de estações na queue.

Caso esperado: O(n) sendo n o número de estações na queue.

**5.2 Remoção de Linha**

Melhor caso: O(1)

Pior Caso: O(n \* m \* p) sendo n o numero de estações em cada horario, m o numero de horarios e p numero de estação para da linha a ser removida.

Caso Esperado: O(n \* m \* p) igual ao pior caso.

**5.3 Consulta das estações de uma linha**

Melhor caso: O(n)

Pior caso: O(n)

Caso Esperado: O(n)

n- número de estações da linha

**5.4 Consulta das linhas de uma estação**

Melhor caso: O(n)

Pior caso: O(n)

Caso Esperado: O(n)

n - número de linhas da estação

**5.5 Inserção de horário**

Melhor caso: O(1)

Pior caso: O( n \* m \* p \* k ) sendo n o número de estações, m tamanho da queue dada, número de horários no respetivo sentido e k número de entries de cada schedule.

**5.6 Remoção de Horário**

Melhor caso: O(1).

Pior Caso: O(nlog(m)) sendo n o número de estações e m o número de horários que passam em cada estação.

Caso esperado: O(nlog(n)) igual ao pior caso.

**5.7 Consulta de horários de uma linha**

Melhor caso: O(n).

Pior caso: O(n)

Caso esperado: O(n)

**5.8 Comboios por estação**

Melhor caso: O(n)

Pior caso: O(n)

Caso Esperado: O(n)

**5.9 Melhor Horário**

Melhor caso: O(n) sendo n tamanho de route

Pior Caso: O((n1+n2)\*m) sendo n1 tamanho da departureSchedules e n2 tamanho da arrivalSchedules

Caso Esperado: O((n1+n2)\*m) igual ao pior caso

**5.10 Terminar aplicação**

Melhor caso: O(T)

Pior caso: O(T)

Caso esperado: O(T)

# **6-Complexidades Espaciais das TADs**

**6.1 TAD RailNetwork**

*private Dictionary*<String,*Line*> lines;

*private Dictionary*<String,*Station*> stations;

lines: (dim(n) + m) sendo n a capacity da tabela de hashing e m número de linhas

stations: (dim(n) + k) sendo n a capacity da tabela de hashing e k número de estações

**6.2 TAD Line:**

*private List*<*Station*> stations;

*private OrderedDictionary*<*Time*, *Schedule*> departureSchedules;

*private OrderedDictionary*<*Time*, *Schedule*> arrivalSchedules;

stations: (n) sendo n o numero de estações da linha

departureSchedules: (n) sendo n o numero de schedules do sentido de partida

arrivalSchedules: (n) sendo n o numero de schedules do sentido contrario

**6.3 TAD Station**

***private OrderedDictionary*<*Line*,*Line*> lines;*// AVL***

***private OrderedDictionary*<Integer,*Time*> trains;**

lines: (n) sendo n numero de linha na estação

trains: (n) sendo n o numero de comboios que passam na estação

**6.4 TAD Schedule**

*private List*<*Entry* <*Station*, *Time*>> route;

route: (n) sendo n o numero de paragens do comboio(Entry<Station,Date>)

**7 Conclusão**

A realização deste projeto serviu para melhorar os nossos conhecimentos sobre as complexidades temporais e espaciais, bem como aprender a melhorar o desempenho de um programa e de saber quais das ED’s fornecidas nas aulas deveremos usar para cada situação.