



NOVA SCHOOL OF
SCIENCE & TECHNOLOGY
DEPARTMENT OF
COMPUTER SCIENCE

RELATÓRIO

PROJETO FINAL

**Sistema de suporte de uma
Rede Ferroviária**

Colaboradores:

Clara Dias (67215, cso.dias@campus.fct.unl)

Gabriela Silva (67286, gt.silva@campus.fct.unl)

Versão I

2024/2025

Índice

1. INTRODUÇÃO.....	3
2. EDs	4
2.1. <i>OrderedDoubleList e SearchableDoubleList.....</i>	<i>5</i>
2.2. <i>SepChainHashtable</i>	<i>6</i>
2.3. <i>AVLTree</i>	<i>7</i>
2.4. <i>TreeSet</i>	<i>8</i>
2.5. <i>VIterator, BSTKeyOrderIterator e TreeSetIterator.....</i>	<i>9</i>
3. TADs.....	10
3.1. <i>Station e StationGet.....</i>	<i>10</i>
3.2. <i>Rail e RailGet</i>	<i>11</i>
3.3. <i>Train e TrainGet</i>	<i>11</i>
3.4. <i>Time</i>	<i>12</i>
3.5. <i>ScheduleEntry.....</i>	<i>12</i>
3.6. <i>PassingTrain.....</i>	<i>12</i>
4. OPERAÇÕES	13
4.1. <i>IL (insertRail ()).....</i>	<i>14</i>
4.2. <i>RL (removeRail ()).....</i>	<i>15</i>
4.3. <i>CL (consultRailStations ()).....</i>	<i>16</i>
4.4. <i>CE (consultStationRails ()).....</i>	<i>17</i>
4.5. <i>IH (insertSchedule ()).....</i>	<i>18</i>
4.6. <i>RH (removeSchedule ()).....</i>	<i>20</i>
4.7. <i>CH (consultRailschedulesDepartingAt ()).....</i>	<i>21</i>
4.8. <i>LC (listStationTrains ()).....</i>	<i>22</i>
4.9. <i>MH (bestSchedule ()).....</i>	<i>23</i>
4.10. <i>TA.....</i>	<i>24</i>
5. COMPLEXIDADE ESPACIAL.....	25
5.1. <i>RailWayNetworkClass</i>	<i>25</i>
5.2. <i>StationClass.....</i>	<i>25</i>
5.3. <i>RailClass.....</i>	<i>26</i>
5.4. <i>TrainClass.....</i>	<i>26</i>
6. CONCLUSÕES E OBSERVAÇÕES	27

1. Introdução

Este documento trata-se de um relatório do trabalho prático da UC **Algoritmos e Estrutura de Dados** (AED) do 2º ano da Licenciatura em Engenharia Informática na Faculdade de Ciências e Tecnologias da UNL.

Neste relatório serão analisados os tipos de estruturas de dados (**TADs**) utilizados, as estruturas de dados (**ED**) fornecidas pelos docentes da UC, tanto como as **criadas por nós**, as operações implementadas e outras componentes relevantes, no âmbito de um **Sistema de Suporte de uma Rede Ferroviária**.

Este projeto foi implementado na linguagem **Java**. Optou-se pela **versão I** completa do trabalho, em que não é permitido o uso do package `java.util` (à exceção da classe `Scanner`).

2. EDs

Neste tópico abordar-se-á a especificação das **Estruturas de Dados** mais relevantes para a realização deste trabalho. Algumas destas EDs foram-nos fornecidas pelos docentes da disciplina, enquanto outras ou foram completadas durante as aulas práticas ou escritas totalmente por nós. Todos estes TADs e as EDs que os implementam encontram-se no pacote *dataStructures* da *source* do programa.

2.1. OrderedDoubleList e SearchableDoubleList

As EDs *OrderedDoubleList* e *SearchableDoubleList* implementam os TADs *Dictionary* e *List*, respetivamente. Intuitivamente percebe-se que uma *SearchableDoubleList* é uma extensão de uma *DoubleList* comum e que uma *OrderedDoubleList* é uma *DoubleList*, só que ordenada por chave.

Uma *SearchableDoubleList* tem uma modificação interessante relativamente a uma *DoubleList* comum: o método *findEquals()*. Tem como função iterar a lista e procura pelo elemento passado no argumento, retornando-o; caso o elemento não exista na lista, retorna *null*.

Operação	Melhor caso	Pior caso	Caso esperado
<i>E</i> <i>findEquals (E)</i>	<i>O(1)</i>	<i>O(n)</i>	<i>O(n)</i>

Tabela 1 – Complexidade temporal de *findEquals ()* de uma SDL (n: número de elementos)

Uma *OrderedDoubleList* é uma implementação de um dicionário ordenado sem o uso de uma tabela de dispersão. Foi-nos indicada esta forma de ordenar uma lista de forma eficiente na primeira fase deste trabalho e mantivemos o seu uso. A característica mais interessante desta ED é a inserção e remoção de elementos, visto que se pretende manter os elementos ordenados pela chave.

Operação	Melhor caso	Pior caso	Caso esperado
<i>V</i> <i>find (K)</i>	<i>O(1)</i>	<i>O(n)</i>	<i>O(1)</i>
<i>V</i> <i>insert (K, V)</i>	<i>O(1)</i>	<i>O(n)</i>	<i>O(1)</i>
<i>V</i> <i>remove (K)</i>	<i>O(1)</i>	<i>O(n)</i>	<i>O(1)</i>

Tabela 2 – Complexidades temporais de métodos de uma ODL (n: número de elementos)

2.2. SepChainHashtable

A ED **SepChainHashtable** estende a implementação de uma **Hashtable** comum que implementa um TAD **Dictionary**. Uma **SepChainHashtable** resolve o problema das colisões de uma **Hashtable** comum da seguinte forma: cada posição da tabela armazena uma lista de elementos associado ao índice dessa posição. Não garante ordenação, mas é eficiente em pesquisas.

Operação	Melhor caso	Pior caso	Caso esperado
$v \text{ } \texttt{find}(k)$	$O(1)$	$O(n)$	$O(1)$
$v \text{ } \texttt{insert}(k, v)$	$O(1)$	$O(n)$	$O(1)$
$v \text{ } \texttt{remove}(k)$	$O(1)$	$O(n)$	$O(1)$

Tabela 3 – Complexidades temporais de métodos de uma HT (n : número de elementos)

2.3. *AVLTree*

Por definição, uma *AVLTree* é uma árvore binária de pesquisa equilibrada. A diferença entre as alturas das subárvore esquerda e direita deve ser 1, no máximo. Numa *AVL*, a verificação desta condição deve ser de **fácil manutenção**. A fim de manter o equilíbrio da árvore, os nós guardam a altura da árvore cuja raiz é o próprio nó.

A utilização de *AVLTrees* no projeto foi bastante útil, visto que os seus métodos de pesquisa, inserção e remoção têm **melhor complexidade temporal** que árvores comuns e outras EDs, mantendo a ordem dos objetos.

Operação	Melhor caso	Pior caso	Caso esperado
$V \text{ } find(K)$	$O(1)$	$O(\log n)$	$O(\log n)$
$V \text{ } insert(K, V)$	$O(1)$	$O(\log n)$	$O(\log n)$
$V \text{ } remove(K)$	$O(1)$	$O(\log n)$	$O(\log n)$

Tabela 4 – Complexidade temporal dos métodos de uma *AVLTree* (n : número de elementos)

Para a implementação correta dos métodos `insert()` e `remove()` da *AVLTree*, recorreu-se a métodos auxiliares da ED **BST** (*BinarySearchTree*): `linkSubtreeInsert()` e `linkSubtreeRemove()`.

2.4. *TreeSet*

Vista a necessidade de ordenar Comboios que passam numa Estação por Hora de passagem, recorreu-se à implementação da ED *TreeSet*. Esta ED tem uma implementação bastante similar a uma *AVLTree*. Em vez de armazenar pares valor-chave, guarda-se apenas **um elemento genérico**.

Um *TreeSet* recebe um **comparador** como argumento, que compara os elementos que o *TreeSet* carrega. Deste modo, o *TreeSet* ordena os elementos segundo esse comparador. Esta ED é útil pela sua simplicidade temporal.

Operação	Melhor caso	Pior caso	Caso esperado
$v \text{ } find(K)$	$O(1)$	$O(\log n)$	$O(\log n)$
$v \text{ } insert(K, V)$	$O(1)$	$O(\log n)$	$O(\log n)$
$v \text{ } remove(K)$	$O(1)$	$O(\log n)$	$O(\log n)$

Tabela 5 – Complexidade temporal dos métodos de um *TreeSet* (n : número de elementos)

Foi necessária, então, a criação da interface ***Comparator***. A partir desta interface é possível comparar qualquer tipo de objeto, consoante a implementação desejada. *Comparator* dispõe um simples método *compare()*, que compara dois objetos segundo as características específicas desse objeto.

2.5. *VIterator*, *BSTKeyOrderIterator* e *TreeSetIterator*

Dada a elevada necessidade de iterar pares chave-valor, recorreu-se à criação de um iterador de valores *VIterator*. Esta ED recebe, simplesmente, um iterador do TAD *Dictionary* e constrói um iterador que itera, apenas, os **valores** dos pares chave-valor do dicionário.

O *BSTKeyOrderIterator* é um iterador que percorre uma árvore de pesquisa binária por ordem **crescente**, i.e., recorre ao percurso **infixo**. Existem outros dois percursos possíveis (prefixo e sufixo) que não foram considerados necessários para este trabalho.

Face à implementação de um *TreeSet*, foi necessária, também, a criação de um iterador que itere um *TreeSet*. O *TreeSetIterator* percorre o *TreeSet* também por ordem **crescente** (percurso infixo) e a sua implementação é bastante semelhante a um *BSTKeyOrderIterator*, com a diferença de guardar elementos singulares genéricos e não pares chave-valor.

3. TADs

Esta secção descreve os **TADs** utilizados neste projeto. Um TAD é uma abstração que descreve o comportamento de um conjunto de dados e as operações que podem ser realizadas por esses dados, i.e., não detalha a sua implementação.

No contexto de um Sistema de Suporte de uma Rede Ferroviária, foi considerada importante a criação dos TADs **Station** (e **StationGet**), **Rail** (e **RailGet**), **Train** (e **TrainGet**), **Time**, **ScheduleEntry** e **PassingTrain**.

3.1. Station e StationGet

Os TADs **Station** e **StationGet**, de uma modo geral, representam uma **Estação**. Neste contexto, uma Estação pode pertencer a várias linhas, permitindo que Comboios passem por ela segundo um determinado horário. Uma Estação pode, ainda, ser uma Estação terminal de uma linha.

O TAD **StationGet** contém apenas a informação relevante sobre uma Estação (nome e número de Linhas a que pertence), enquanto o TAD **Station** possui, para além disso, as operações que manipulam as propriedades dele próprio (inserções e remoções de Linhas, Comboios, etc).

Uma Estação armazena as Linhas a que pertence numa **OrderedDoubleList** e os Comboios que passam por ela num **TreeSet**, que guarda um par de dados Comboio-Hora (de passagem).

3.2. *Rail* e *RailGet*

Os TADs *Rail* e *RailGet* desenham uma **Linha** de Comboios. Uma Linha é constituída por um conjunto de Estações que representam o percurso de um Comboio, sendo a inicial e a final Estações terminais.

De forma semelhante ao TAD *StationGet*, o *RailGet* contém apenas a informação essencial (nome e quantidade de Estações que contém). Por sua vez, o TAD *Rail* abrange operações que manipulam as propriedades deste TAD (inserções e remoções de Comboios, etc).

Uma Linha guarda a sequência das suas Estações numa *SearchableDoubleList* e os Comboios que fazem o seu percurso em duas *AVLTrees*, uma para cada sentido da rota, ambas ordenadas pela Hora de partida da respetiva Estação de partida. Para além disso, todos os Comboios são, ainda, guardados numa *DoubleList* comum, i.e., desordenada, vista a necessidade de remover todos os Comboios da Linha quando esta é apagada do sistema.

3.3. *Train* e *TrainGet*

O TAD *Train* diz respeito a um **Comboio**. Um Comboio, neste contexto, respeita um determinado **horário** – conjunto de entradas de pares Estação-Tempo (*ScheduleEntry*) – passando por uma sequência de Estações de uma Linha. Cada Comboio tem um número que serve como identificador único.

Tal como *StationGet* e *RailGet*, o TAD *TrainGet* apresenta, também, apenas a informação essencial ao utilizador (número e lista de *ScheduleEntries*), enquanto o TAD *Train* contém informações mais relevantes para o sistema (Estações terminais e Hora a que o Comboio passa por elas).

O conjunto de entradas de pares Estação-Tempo (TAD *ScheduleEntry*) é guardado neste TAD numa *DoubleList* comum.

3.4. Time

O TAD **Time** guarda os dados que representam uma hora no formato $hh:mm$ – inteiro para horas e inteiro para minutos. Este TAD tem um *compareTo* implementado para ser possível listar um conjunto de objetos **Time** segundo uma ordem específica.

3.5. ScheduleEntry

O TAD **ScheduleEntry** tem como função guardar um par de dados **Estação-Hora**. Uma *ScheduleEntry* é uma entrada de um horário, i.e., é um par de dados Estação-Hora de um horário. Um **horário** é, então, uma lista de *ScheduleEntries*. Um Comboio respeita um horário, passando por cada Estação à Hora associada ao par, portanto o TAD **Train** tem uma lista de *ScheduleEntries*.

Como o *input* só é analisado após um duplo *enter* e não a cada *line break*, durante a tarefa de se inserir um horário num Comboio é guardada uma lista de *ScheduleEntries* e, **após validada**, o horário é inserido no Comboio, que é, finalmente, criado.

3.6. PassingTrain

O TAD **PassingTrain** armazena outro par de dados: **Comboio-Hora**. Tendo em conta que é necessário ordenar a passagem de Comboios numa determinada Estação por hora de passagem, e, no caso de empate, por ordem crescente de número dos Comboios, é importante o armazenamento desse par de dados. Uma Estação tem um *TreeSet* de *PassingTrain* para esse efeito.

4. Operações

Esta secção descreve as operações que foram implementadas para o funcionamento do programa. Analisa-se em ponto forte as **complexidades temporais** destas operações. São descritos os seguintes comandos:

- ***IL***: Inserir uma Linha no sistema;
- ***RL***: Remover uma Linha do sistema;
- ***CL***: Consultar as Estações de uma Linha;
- ***CE***: Consultar as Linhas a que uma Estação pertence;
- ***IH***: Inserir um horário numa Linha;
- ***RH***: Remover um horário de uma Linha;
- ***CH***: Consultar horários de uma Linha;
- ***LC***: Listar os Comboios que passam numa Estação;
- ***MH***: Melhor horário de um percurso;
- ***TA***: Terminar a aplicação.

Vista a utilização recorrente de alguns métodos auxiliares nas classes importantes do projeto, segue-se uma tabela do essencial desses métodos.

Método	Função	Complexidade Temporal
<code>getRail()</code> <code>getStation()</code>	<code>find()</code> <i>SepChainHashtable</i>	$O(1)$ (melhor e esperado) $O(n)$ (pior)
<code>findSchedule()</code>	<code>find()</code> <i>AVLTree</i>	$O(1)$ (melhor) $O(\log n)$ (pior e esperado)
<code>isTerminalStation()</code>	<code>equals()</code> <i>StationClass</i>	$O(1)$ (todos)

Tabela 6 – Função e complexidade temporal de métodos recorrentes (n : número de elementos).

4.1. IL (*insertRail()*)

Dado um nome e um conjunto de nomes de Estações, este método **insere uma Linha** no sistema. A Linha é inserida no sistema apenas se esta não existir no sistema, i.e., se não existir uma Linha com o nome dado no sistema. Para verificar esta condição, o sistema recorre ao método *getRail()*.

A simples tarefa de inserir a Linha no sistema é feita através do *insert()* de uma *SepChainHashtable*. No entanto, as suas Estações são inseridas tanto na própria Linha como no sistema e a Linha é, ainda, inserida em cada Estação. É verificado, ainda, se alguma das Estações da Linha já existe no sistema: caso não exista, uma nova Estação é criada e inserida no sistema; caso contrário, encontra-se a Estação segundo o método *getStation()* e esta é inserida na Linha. Esta tarefa é realizada pelo método privado *addStationsToRail()* da classe de topo, que chama o *insert()* de uma *SepChainHashtable* e o *addLast()* de uma *OrderedDoubleList* para **cada** Estação da Linha.

Sendo assim, a complexidade temporal do comando IH dependerá da complexidade temporal do método *addStationsToRail()*. Os pior e esperado casos são ambos **$O(n)$** , onde n é o número de Estações da Linha. O melhor caso acontece, então, quando o número de Estações da Linha é mínimo ($n = 2$).

	Operação	Melhor	Pior	Esperado
(Main)	<i>insertRail</i>	$O(n)$ ($n = 2$)	$O(n)$	$O(n)$
(Classe sistema)	<i>addStationsToRail</i>	$O(n)$ ($n = 2$)	$O(n)$	$O(n)$

Tabela 7 – Complexidade temporal do comando *insertRail()*

4.2. RL (`removeRail()`)

Dada uma Linha, este método **remove-a** do sistema. A Linha é removida se e só se já existir no sistema. Esta verificação é feita segundo o `getRail()`. Novamente, a simples tarefa de remover a Linha do sistema é feita pelo método `remove()` da *SepChainHashtable*.

Quando uma Linha é removida do sistema:

- A Linha é removida de **cada Estação** a que lhe pertencia;
- Se uma Estação apenas fizer parte dessa Linha, esta Estação é **removida automaticamente** do sistema;
- Todos os Comboios (horários) inseridos nesta Linha são apagados.

A complexidade do comando RL depende, então, dos métodos `removeRailFromStations()` e `removeAllTrainsFromStations()` (classe de topo e TAD *Rail*, respetivamente).

`removeRailFromStations()` percorre todas as Estações da Linha e remove a Linha das estações. A sua complexidade temporal baseia-se, então em **$O(n)$** , sendo n o número de Estações da Linha, em que o melhor caso ocorre quando n é mínimo ($n = 2$).

`removeAllTrainsFromStations()` percorre todos as *ScheduleEntries* de todos os Comboios e remove os Comboios de cada Estação associada à *ScheduleEntry*. A sua complexidade temporal será, então, **$O(c*h)$** , sendo c o número de Comboios da Linha e h o número máximo de *ScheduleEntries* de cada Comboio. O melhor caso acontece quando a Linha não tem qualquer Comboio inserido, **$O(1)$** .

(Main)	Operação	Melhor	Pior	Esperado
(Classe sistema)	<code>removeRail</code>	$O(n)$ ($n = 2$)	$O(n + c*h)$	$O(n + c*h)$
(TAD <i>Rail</i>)	<code>removeRailFromStations</code>	$O(n)$ ($n = 2$)	$O(n)$	$O(n)$
	<code>removeAllTrainsFromStations</code>	$O(1)$	$O(c*h)$	$O(c*h)$

Tabela 8 – Complexidade temporal do comando `removeRail()`

4.3. CL (*consultRailStations()*)

Dado o nome da Linha, este método **lista todas as Estações que pertencem à Linha**. *listRailsStations()* retorna um iterador (TAD *List*) de Estações. As Estações são iteradas por ordem sequencial de inserção. O comando é realizado com sucesso se a Linha existir no sistema. Verifica-se esta condição segundo o método *getRail()*.

Este método não depende especialmente de outro método que altere significativamente a sua complexidade temporal que é, para os pior e esperado casos, **$O(n)$** , onde n é o número de estações da linha. O melhor caso possível acontece quando o número de Estações da Linha é mínimo ($n = 2$).

listRailsStations() retorna apenas o iterador vindo do método *listStations()* do TAD *Rail*. *listStations()* pode ter complexidade temporal **$O(n)$** se o iterador tiver de ser retornado por ordem inversa mas, neste comando, não é necessário. Logo, a complexidade temporal é linear, **$O(1)$** .

	Operação	Melhor caso	Pior caso	Caso esperado
(Main)	<i>consultRailStations</i>	$O(n)$ ($n = 2$)	$O(n)$	$O(n)$
(Classe sistema)	<i>listRailsStations</i>	$O(1)$	$O(1)$	$O(1)$
(TAD Rail)	<i>listStations</i>	$O(1)$	$O(1)$	$O(1)$

Tabela 9 – Complexidade temporal do comando *consultRailStations()*

4.4. CE (*consultStationRails()*)

Dada uma Estação, o método *listStationRails()* **lista todas as Linhas a que a Estação pertence** através do método *listRails()* (TAD *Station*). A listagem é feita por ordem lexicográfica. Este método devolve um *VIterator* (Iterador de valores) de um *Dictionary*, que contém apenas os objetos Linha.

Este método tem complexidade temporal, nos pior e esperado casos, $O(n)$, onde n é o número de Linhas da Estação, e, no melhor caso, $O(1)$, quando a Estação pertence apenas a uma Linha.

	Operação	Melhor caso	Pior caso	Caso esperado
(Main)	<i>consultStationRails</i>	$O(1)$	$O(n)$	$O(n)$
(Classe sistema)	<i>listStationRails</i>	$O(1)$	$O(1)$	$O(1)$
(TAD <i>Station</i>)	<i>listRails</i>	$O(1)$	$O(1)$	$O(1)$

Tabela 10 – Complexidade temporal do método *consultStationRails()*

4.5. IH (*insertSchedule()*)

São fornecidos a Linha, o número do Comboio e uma lista de pares de Estações e Horas que compõem o respetivo horário do Comboio. O processo de **inserir um horário** só acontece se a Linha existir no sistema. Esta verificação é feita pelo método *getRail()*.

A validação do horário a ser inserido é feita pelo método *validateSchedule()*. Um horário é válido quando respeita todas as condições:

- A primeira Estação inserida é uma das Estações terminal da Linha;
- O horário foi inserido por ordem crescente de Horas;
- A sequência de Estações respeita a ordem da rota da Linha, não tendo necessariamente de passar por todas as Estações da Linha;
- O Comboio respetivo ao horário a ser inserido não ultrapassa outro Comboio existente na Linha. Esta condição é verificada pelo método auxiliar *isOvertakingAvoided()*.

Quando o horário é validado, o respetivo TAD *Train* é criado e inserido consoante o seu sentido (normal ou contrário).

A complexidade temporal deste comando depende, então, dos métodos *validateSchedule()* e *isOvertakingAvoided()*.

validateSchedule() percorre todas as *ScheduleEntries* do horário e todas as Estações da Linha, portanto os pior e esperados casos da complexidade temporal deste método são $O(n * m)$, onde n é o número de Estações da Linha e m é o número de *ScheduleEntries* do horário a ser inserido. O melhor caso acontece quando n e m são mínimos ($n = m = 2$).

isOvertakingAvoided() percorre todos os Comboios que partam da mesma Estação de partida que o novo horário a ser inserido e, para cada Comboio desses, percorre também as *ScheduleEntries* do seu horário e, ainda, as *ScheduleEntries* do novo horário a ser inserido. Os pior e esperados casos são, portanto, $O(c * m * h)$, onde c é o número de Comboios que parte da mesma Estação

que o novo horário a ser inserido e h é o número máximo de *ScheduleEntries* do Comboio a ser analisado. O melhor caso acontece quando não parte nenhum Comboio da mesma Estação que o novo horário a ser inserido, $c = 0$, $O(1)$.

	Operação	Melhor caso	Pior caso	Caso esperado
(Main)	<i>insertSchedule</i>	$O(n * m) \ (n=m=2)$	$O(n * m + c * m * h)$	$O(n * m + c * m * h)$
(Classe sistema)	<i>validateSchedule</i>	$O(n * m) \ (n=m=2)$	$O(n * m)$	$O(n * m)$
(Classe sistema)	<i>isOvertakingAvoided</i>	$O(1)$	$O(c * m * h)$	$O(c * m * h)$

Tabela 11 – Complexidade temporal do método *insertSchedule()*

4.6. RH (`removeSchedule()`)

Dada uma Linha, uma Estação e uma Hora de partida, o Comboio que respeita o horário que parte nessa Estação a essa Hora é **removido** do sistema. É verificada a existência tanto da Linha como do horário, segundo os métodos `getRail()` e `findSchedule()`.

Quando um horário é removido do sistema, este tem de ser removido tanto da Linha como também de cada Estação em que passa. Para tal, os métodos `removeTrain()` e `removeTrainFromStations()` são utilizados (classe sistema e TAD *Rail*, respetivamente), portanto a complexidade temporal deste comando depende desses métodos.

`removeTrain()` apenas remove diretamente o horário (Comboio que respeita esse horário) do conjunto de Comboios da Linha, dependendo do sentido do percurso do horário, i.e., dependendo da Estação de partida do horário. Usa o `remove()` de uma *AVLTree*. Neste caso, todos os casos são **$O(1)$** , mas o método auxiliar `removeTrainFromStations()` é, ainda, chamado dentro de `removeTrain()`.

`removeTrainFromStations()` percorre todas as Estações da Linha e remove o Comboio a ser removido da Linha. A sua complexidade temporal será, então, **$O(n)$** , onde n é o número de Estações da Linha. O melhor caso acontece quando n é mínimo ($n = 2$).

	Operação	Melhor caso	Pior caso	Caso esperado
(Main)	<code>removeSchedule</code>	$O(1)$	$O(1)$	$O(1)$
(Classe sistema)	<code>removeTrain</code>	$O(1)$	$O(1)$	$O(1)$
(TAD <i>Rail</i>)	<code>removeTrainFromStations</code>	$O(n)$ ($n=2$)	$O(n)$	$O(n)$

Tabela 12 – Complexidade temporal do método `removeSchedule()`

4.7. CH (*consultRailSchedulesDepartingAt()*)

É dada uma Linha e uma Estação de partida. Após a verificação da existência de ambas, segundo os métodos *getRail()*, *getStation()* e *isTerminal()*, o método *consultRailSchedule()* devolve um *VIterator* de **Comboios dessa Linha que partam dessa Estação de partida**, recorrendo ao método *listTrainsDepartingAt()* (TAD *Rail*).

O comando em si lista cada *ScheduleEntry* de cada horário que cada Comboio respeita dentro dessa Linha, i.e., a cada Comboio que parte da Estação de partida, percorre a lista de *ScheduleEntries* do horário desse Comboio, segundo o método *listScheduleEntries()* (TAD *Train*).

Todos os métodos auxiliares deste comando têm complexidade temporal linear, visto apenas retornarem iteradores. O próprio comando é quem percorre os iteradores, portanto a sua complexidade temporal será $O(n*m)$, onde n é o número de Comboios da Linha que partem da Estação de partida dada e m é o número máximo de *ScheduleEntries* do horário de cada Comboio. O melhor caso acontece quando a Linha não tem Comboios nenhum que partam dessa Estação de partida, $n = m = 0$, $O(1)$.

	Operação	Melhor caso	Pior caso	Caso esperado
(Main)	<i>consultRailSchedulesDepartingAt</i>	$O(1)$	$O(n*m)$	$O(n*m)$
(Classe sistema)	<i>consultRailSchedule</i>	$O(1)$	$O(1)$	$O(1)$
(TAD <i>Rail</i>)	<i>listTrainsDepartingAt</i>	$O(1)$	$O(1)$	$O(1)$
(TAD <i>Train</i>)	<i>listScheduleEntries</i>	$O(1)$	$O(1)$	$O(1)$

Tabela 13 – Complexidade temporal do método *consultRailSchedulesDepartingAt()*

4.8. LC (*listStationTrains()*)

Para este comando é fornecida uma Estação. A existência desta Estação no sistema é verificada segundo o método *getStation()*. O comando **lista todos os Comboios que passam nessa Estação, por ordem de passagem**, i.e., por Hora de passagem na Estação.

Os métodos auxiliares deste comando apenas retornam um iterador, portanto a complexidade temporal desses métodos será sempre linear, $O(1)$. Como referido, retornam um iterador ordenado por Hora de passagem.

O comando em si percorre todos os Comboios (TAD *PassingTrain*) que passam nessa Estação e imprime o seu número e a Hora de passagem. Logo, a sua complexidade temporal será $O(n)$ nos casos pior e esperado, onde n é o número de Comboios que passam pela Estação. O melhor caso acontece quando não passa nenhum Comboio na Estação, i.e., $n = 0$, $O(1)$.

	Operação	Melhor caso	Pior caso	Caso esperado
(Main)	<i>listStationTrains</i>	$O(1)$	$O(n)$	$O(n)$
(Classe sistema)	<i>listTrains</i>	$O(1)$	$O(1)$	$O(1)$

Tabela 14 – Complexidade temporal do método *listStationTrains()*

4.9. MH (*bestSchedule()*)

São fornecidos uma Linha, duas Estações da Linha e uma Hora. As estações são, respetivamente, o ponto de partida e o ponto de chegada de um percurso dentro da Linha. A Hora é a Hora expectável de chegar ao ponto de chegada. Este comando **lista o melhor horário que inclui este percurso que chegue até à Hora expectável.**

As estações não têm que necessariamente ser terminais, mas têm de pertencer à Linha. Como sempre, a existência da Linha e das Estações é verificada segundo os métodos *getRail()* e *getStation()*.

O melhor horário é calculado segundo o método *getBestSchedule()* (TAD *Rail*). Este método percorre cada Comboio (horário) da Linha que passa no mesmo sentido que o percurso dado e, para cada Comboio, percorre cada *ScheduleEntry* do horário que o Comboio respeita. Logo, a complexidade temporal deste método é $O(c * h)$, onde c é o número de Comboios da Linha nesse sentido e h é o número máximo de *ScheduleEntries* do horário do Comboio. O melhor caso acontece quando não existem Comboios a atuar nesse sentido, i.e., $c = 0$, $O(1)$.

O comando em si percorre todas as entradas do melhor horário calculado. Logo, a sua complexidade temporal nos pior e esperado casos é $O(c * h * n)$, sendo n o número de entradas do melhor horário encontrado e, no melhor caso, $O(1)$, quando não existem Comboios a atuar no sentido do percurso dado, i.e., $c = 0$, $O(1)$.

	Operação	Melhor caso	Pior caso	Caso esperado
(Main)	<i>bestSchedule</i>	$O(1)$	$O(c * h * n)$	$O(c * h * n)$
(Classe sistema)	<i>getBestTrain</i>	$O(1)$	$O(c * h)$	$O(c * h)$
(TAD <i>Rail</i>)	<i>getBestSchedule</i>	$O(1)$	$O(c * h)$	$O(c * h)$

Tabela 15 – Complexidade temporal do método *bestSchedule()*

4.10. TA

O comando TA termina a aplicação.

Operação <small>(Main)</small>	Melhor caso	Pior caso	Caso esperado
<code>exit</code>	$O(1)$	$O(1)$	$O(1)$

Tabela 16 – Complexidade temporal do método `exit ()`

5. Complexidade Espacial

Neste tópico é analisada a **complexidade espacial** do programa, bem como a justificação do planeamento das estruturas de dados a que se recorreu.

5.1. RailWayNetworkClass

A classe **RailWayNetworkClass** é a classe sistema do programa. Por este motivo, é necessário armazenar toda a informação sobre os TADs *Station* e *Rail* nesta classe. Como a pesquisa é um fator essencial, foram implementados dois TAD *Dictionary*. Tem-se, então, duas *Hashtables* (*SepChainHashtable*): uma de Estações e outra de Linhas.

Assim sendo, fala-se numa complexidade espacial de $O(l) + O(e)$, onde l e e são os números de Linhas e de Estações, respetivamente.

5.2. StationClass

A classe **StationClass** implementa os métodos do TAD *Station*. Numa Estação é importante armazenar os TADs individuais *Rail* e *Train*. Para tal, foram implementados um TAD *Dictionary* por uma ED *OrderedDoubleList*, e um TAD *TreeSet* por uma ED *TreeSetClass*. A escolha destas EDs baseia-se nas necessidades da ordenação lexicográfica do TAD *Rail* e da ordenação por Hora de passagem do TAD *Train*.

Em termos de complexidade espacial, tem-se $O(l) + O(c)$, onde l e c são os números de Linhas e de Comboios, respetivamente.

5.3. *RailClass*

A classe *RailClass* implementa os métodos do TAD *Rail*. Uma Linha requer o armazenamento dos TADs *Station* e *Train*. Sendo assim, foram implementados um TAD *SearchableDoubleList* implementado pela ED *SearchableDoubleListClass*, para armazenar o conjunto do TAD *Station* da Linha. Já o TAD *Train* é guardado em dois TAD *OrderedDictionary*, implementados por *AVLTree*, vista a necessidade de separar os Comboios pela direção da rota da Linha que respeitam, ordenados por Hora de partida. Foi, ainda, implementado um TAD *List* por uma *DoubleList* que carrega todos os Comboios que atuam na Linha, desordenados.

Embora algumas destas escolhas de EDs prejudiquem a complexidade espacial, estas são justificadas pela necessidade de manter equilíbrio entre as complexidade espacial e temporal. Logo, são mais eficientes noutras componentes.

Tem-se, então, uma complexidade espacial traduzida por $O(c_1) + O(c_2) + O(c) + O(e)$, onde c_1 e c_2 não os números de Comboios que atuam na direção comum e na direção inversa, respetivamente, c é a soma c_1+c_2 e e é o numero de Estações da Linha.

5.4. *TrainClass*

Na classe *TrainClass* viu-se a necessidade de guardar o TAD individual *ScheduleEntry*. Deste modo, foi usada um TAD *List*, implementado pela ED *DoubleList*. Como não é necessário nem ordenar as entradas nem pesquisar por entradas, justifica-se a escolha desta ED, embora a sua complexidade espacial.

Sabe-se, então, que se tem uma complexidade espacial dada por $O(s)$, sendo s o número máximo de *ScheduleEntries* que compõe o horário que o TAD *Train* respeita.

6. Conclusões e observações

Todos os conhecimentos aplicados neste trabalho prático foram adquiridos das UCs Introdução à Programação (**IP**), Programação Orientada a Objetos (**POO**) e Algoritmos e Estruturas de Dados (**AED**), do curso de Engenharia Informática da FCT NOVA.

O projeto em si foi bem sucedido, tendo pontuação 100 no *Mooshak* em ambas a fases de desenvolvimento e entrega. A aplicação prática de várias EDs e TADs de forma eficiente foi crucial para o sucesso deste trabalho. As colaboradoras trabalharam sempre em conjunto, havendo sempre espaço para discussões e partilha de opiniões, sempre com o mesmo fim.

Embora se tenham atingido os objetivos principais, não deixa de haver espaço para melhorias. É, também, de se notar a versatilidade da implementação, que deixa em aberto a criação de novas funcionalidades e limitações no programa que foi desenhado. Por exemplo, futuramente poder-se-á limitar tanto o tempo de passagem de um Comboio como o número de Linhas que pertencem a uma Estação, de modo a tornar o problema mais realista.