

**Algoritmos e Estruturas de Dados – 2021/2022**

**1º Semestre**

**Relatório Projeto BdFI**

**(Base de Dados de Filmes na Internet)**

**Realizado por:**

**60236, Guilherme Poças**

**61052, João Oliveira**

Índice

[1 Introdução 2](#_Toc90572062)

[2 Estrutura do projeto 3](#_Toc90572063)

[2.1 Diagrama de classes (package BdFI) 3](#_Toc90572064)

[2.2 Tipos Abstratos de Dados 4](#_Toc90572065)

[2.2.1 Database 4](#_Toc90572066)

[2.2.2 Show 5](#_Toc90572067)

[2.2.3 Person 6](#_Toc90572068)

[2.2.4 Participation 6](#_Toc90572069)

[2.3 Descrição das operações e estudo das complexidades temporais 7](#_Toc90572070)

[2.3.1 Adicionar profissional 7](#_Toc90572071)

[2.3.2 Adicionar filme ou programa 7](#_Toc90572072)

[2.3.3 Inserir participação de profissional em programa 8](#_Toc90572073)

[2.3.4 Estreia de programa 8](#_Toc90572074)

[2.3.5 Remover programa 9](#_Toc90572075)

[2.3.6 Categorizar programa 10](#_Toc90572076)

[2.3.7 Consultar dados de programa 10](#_Toc90572077)

[2.3.8 Avaliar programa 11](#_Toc90572078)

[2.3.9 Consultar dados de profissional 11](#_Toc90572079)

[2.3.10 Listar programas associados a um profissional 12](#_Toc90572080)

[2.3.11 Listar participações em programas 12](#_Toc90572081)

[2.3.12 Listar os melhores programas 13](#_Toc90572082)

[2.3.13 Listar programas com avaliação 13](#_Toc90572083)

[2.3.14 Listar programas com palavra-chave (*tag*) 14](#_Toc90572084)

[2.3.15 Terminar execução 14](#_Toc90572085)

[3 Complexidade espacial 15](#_Toc90572086)

[4 Conclusão 16](#_Toc90572087)

# Introdução

Durante o 1º semestre do 2ª ano, na disciplina de Algoritmos e Estruturas de Dados, abordamos, entre outros assuntos, complexidades temporais e estruturas de dados. A fim de colocarmos em prática os temas aprendidos, foi-nos pedido para fazer um projeto em duas fases. Uma primeira fase em que o programa teria muitos poucos dados e numa segunda fase em que tivesse preparado para ter milhares de dados e funcionar de forma eficiente.

Neste relatório vamos falar sobre as estruturas de dados utilizadas e a justificação de a termos usado. Iremos também considerar as complexidades temporais das várias operações que o nosso programa suporta bem como a complexidade espacial do programa na sua totalidade.

# Estrutura do projeto

Neste capítulo está o diagrama de classes do nosso projeto e também uma explicação das estruturas de dados usadas.

## Diagrama de classes (package BdFI)

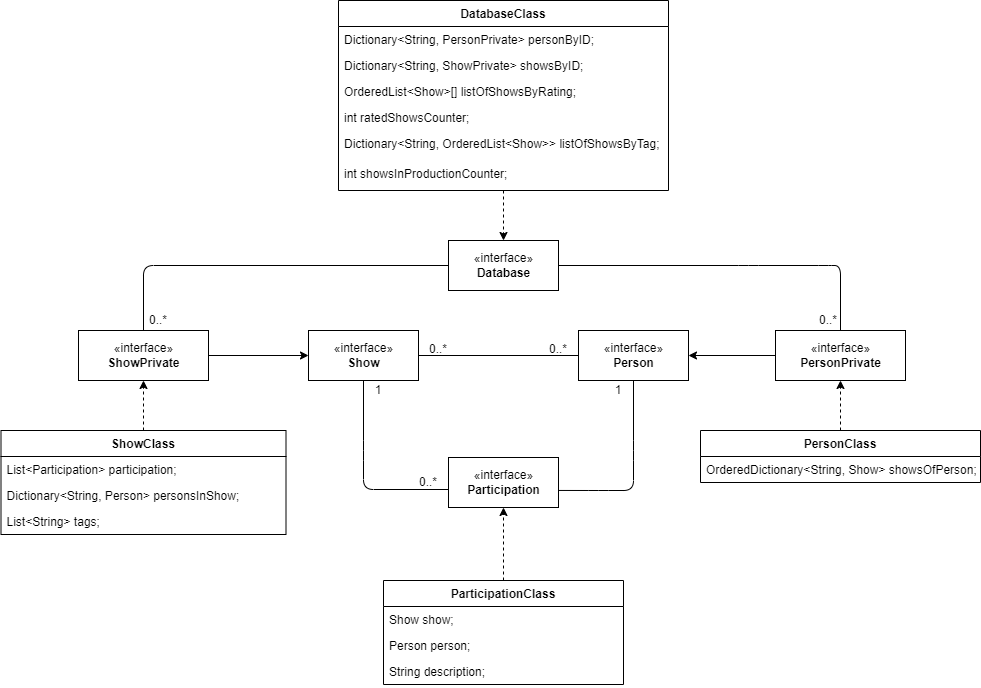


Figura - Diagrama de Classes

Na imagem acima encontra-se um diagrama com as classes e interfaces do nosso programa, com as estruturas de dados presentes em cada uma.

## Tipos Abstratos de Dados

### Database

A *Database* é a interface que é implementada pela *DatabaseClass*. Lá definimos 4 estruturas de dados e duas variáveis auxiliares:

1. *Dictionary<String, PersonPrivate> personByID;*
2. *Dictionary<String, ShowPrivate> showsByID;*
3. *OrderedList<Show>[] listOfShowsByRating;*
4. *Dictionary<String, OrderedList<Show>> listOfShowsByTag;*
5. *int ratedShowsCounter;*
6. *int showsInProductionCounter;*

Tanto na 1ª como 2ª, as implementações são *SepChainHashTable*. Optámos por ela pois não precisamos de manter a ordem de inserção dos elementos, e tanto a pesquisa como a remoção têm complexidade constante. A chave é o *id* da pessoa e do programa, respetivamente, e o valor é a *personPrivate* e o *showPrivate*, respetivamente.

O mapa dos programas (*showsByID*) é inicializado com um tamanho de 200. Este tamanho permite que sejam feitos poucos redimensionamentos da tabela de dispersão. No máximo serão feitos 4, assumindo que não existirão mais de 5000 programas.

O mapa das pessoas (*personByID)* é inicializado com um tamanho de 400. Este tamanho permite que sejam feitos, no máximo, 4 redimensionamentos, assumindo que não existirão mais de 10000 pessoas.

A 3ª é um vetor normal de *OrderedList* que implementam uma *AVLTreeE*. É um vetor com tamanho de 11, sendo que em cada posição estão todos os *shows* com o *rating* correspondente ao índice do vetor. Os *shows* em cada posição estão ordenados por título. Optámos pelo uso do vetor, pois sabemos logo de início o número de classificações, que vão de 0 a 10, e também para não ocupar mais espaço do que o necessário.

Como utilizamos uma variável auxiliar *ratedShowsCounter*, que é incrementada sempre que se dá uma classificação a um programa não classificado. Assim deixa de ser necessário percorrer a estrutura para saber o número de programas classificados.

Na 4ª, à semelhança da 1ª e 2ª, a implementação é um *SepChainHashTable*. A chave é uma *tag*. O valor é uma lista ordenada, implementada por uma *AVLTreeE*, que contém os shows com a *tag* da chave. Os *shows* estão ordenados por título.

Este mapa é inicializado com o tamanho predefinido uma vez que não sabemos qual o número de *tags* que podem existir. Assumimos que será um número reduzido.

Vale a pena notar que também é utilizada a variável *showsInProductionCounter* para verificar se existe algum programa que já acabou a produção. Esta variável é incrementada sempre que se adiciona um programa com o ano corrente, e decrementada quando um programa estreia ou é removido.

### Show

O *Show* representa um programa da Base de Dados, no qual podem participar pessoas e ao qual pode ser atribuído uma avaliação e uma *tag*. É implementado pela *ShowClass*.

Define 3 estruturas de dados:

1. *List<Participation> participation;*
2. *Dictionary<String, Person> personsInShow;*
3. *List<String> tags;*

Na 1ª e 3ª, as implementações serão *DoubleList*, pois na 1ª estrutura é necessário manter os dados por ordem de inserção, para uma futura listagem, e apenas utilizamos a 3ª para fazer um iterador, e inserções no final da lista, que têm ambas complexidade constante.

A 2ª estrutura tem uma implementação *SepChainHashTable*, visto que queremos guardar as pessoas que participam no programa de forma que não existam pessoas repetidas. Por isso, como não queremos adicionar pessoas iguais (com o mesmo *id*), optámos pelo uso desta estrutura que tem complexidade constante na inserção e não terá chaves iguais. A chave é o *id* da pessoa e o valor será a respetiva *Person*.

Este mapa é inicializado com o tamanho predefinido uma vez que só sabemos que uma pessoa participa em centenas de programas. O tamanho predefinido é 50, por isso assumindo que não participará em mais de 842 programas, serão feitos no máximo 3 redimensionamentos.

#### ShowPrivate

O *ShowPrivate* é uma interface que estende o *Show* com os métodos que o modificam de alguma forma, visto que o *Show* apenas terá os métodos que não alteram os dados. Desta forma podemos utilizar o *ShowPrivate* na *Database*, e apenas enviamos o *Show* para fora do *package*, de forma a garantir que não são alterados dados pelo utilizador.

### Person

A *Person* representa um profissional da Base de Dados, que pode participar de alguma forma num programa. É implementado pela *PersonClass*

Define 1 estrutura de dados:

1. *OrderedDictionary<String, Show> showsOfPerson;*

A estrutura é implementada pela *AVLTree*, onde a chave é o *id* do programa e o valor o respetivo *Show*. É usada esta implementação pois é necessário listar os programas de uma pessoa ordenados alfabeticamente pelo seu *id*, e porque a *AVLTree* é uma estrutura de dados bastante eficiente para manter elementos ordenados.

#### PersonPrivate

A *PersonPrivate* é uma interface semelhante ao *ShowPrivate*, onde esta estende a *Person* com os métodos que a modificam, garantido a segurança dos dados fora da *package*.

### Participation

A *Participation* é uma interface que é implementada pela *ParticipationClass*. Contem informações relativamente a uma participação de uma pessoa num programa.

Guarda a pessoa, o programa e uma descrição da participação da pessoa nesse programa.

## Descrição das operações e estudo das complexidades temporais

### Adicionar profissional

Um profissional é adicionado à base de dados, sendo apenas necessário verificar que o profissional não existe no dicionário *personByID*, e caso não exista, criamos um novo objeto e inserimo-lo.

|  |  |  |  |
| --- | --- | --- | --- |
| Operação | Melhor Caso | Pior Caso | Caso Esperado |
| Pesquisa do profissional | O(1) | O(*n1*) | O(1) |
| Criação do objeto do profissional | O(1) | O(1) | O(1) |
| Inserção do profissional | O(1) | O(*n1*) | O(1) |
| Complexidade total | O(1) | O(*n1*) | O(1) |

Legenda

*n1* – Número total de pessoas.

### Adicionar filme ou programa

É adicionado um filme ou programa à base de dados, verificando que ele não existe no dicionário *showsByID*, e, caso não exista, é criado um objeto novo e depois inserido no dicionário.

Caso o programa tenha o mesmo ano de produção que o ano corrente, também é incrementada a variável *ShowsInProductionCounter.*

|  |  |  |  |
| --- | --- | --- | --- |
| Operação | Melhor Caso | Pior Caso | Caso Esperado |
| Pesquisa do programa | O(1) | O(*n1*) | O(1) |
| Criação do objeto do programa | O(1) | O(1) | O(1) |
| Inserção do programa | O(1) | O(*n1*) | O(1) |
| Complexidade total | O(1) | O(*n1*) | O(1) |

Legenda

*n1* – Número total de programas.

### Inserir participação de profissional em programa

Um profissional participa num certo programa, então procura-se o profissional e o programa, nos respetivos dicionários, *personByID* e *showByID*.

Depois adiciona-se o programa aos programas em que o profissional participou.

Por fim, cria-se o objeto e adiciona-se às participações feitas no programa bem como se adiciona a pessoa que participou ao dicionário de pessoas que participaram num programa.

|  |  |  |  |
| --- | --- | --- | --- |
| Operação | Melhor Caso | Pior Caso | Caso Esperado |
| Procura da pessoa | O(1) | O(*n1*) | O(1) |
| Procura do programa | O(1) | O(*n2*) | O(1) |
| Criação do objeto da Participação | O(1) | O(1) | O(1) |
| Inserção de programa em que a pessoa participou | O(1) | O(log(*n3*)) | O(log(*n3*)) |
| Inserção da participação no programa | O(1) | O(1) | O(1) |
| Inserção da pessoa no programa | O(1) | O(*n4*) | O(1) |
| Complexidade total | O(1) | O(*max*(*n1, n2)*)\* | O(log(*n2*)) |

\* *n1 > n4* e *n2 > n3*

Legenda

*n1* – Número total de pessoas;

*n2* – Número total de programas;

*n3* – Número de programas em que a pessoa participou;

*n4* – Número de pessoas que participaram num programa.

### Estreia de programa

Um dado programa estreia, saindo do estado de produção, logo apenas precisamos

de procurar o programa no dicionário, e alterar o seu estado de produção. A variável auxiliar *showsInProductionCounter* é decrementada uma vez que o programa deixa de estar em produção.

|  |  |  |  |
| --- | --- | --- | --- |
| Operação | Melhor Caso | Pior Caso | Caso Esperado |
| Procura do programa | O(1) | O(*n1*) | O(1) |
| Complexidade total | O(1) | O(*n1*) | O(1) |

Legenda

*n1* – Número total de programas;

### Remover programa

Esta é a operação que terá a maior complexidade pois há a necessidade de remover o programa de várias estruturas de dados.

Uma vez que só se podem remover programas em produção, a variável auxiliar *showsInProductionCounter* é decrementada.

Em primeiro lugar é necessário ver se o programa com o *id* dado existe.

Uma vez que cada pessoa guarda os programas onde participa, ordenados por *id*, é necessário removê-los de lá. Logo há a necessidade de para cada pessoa no programa remover o mesmo.

Depois é necessário remover o programa do vetor que contêm os programas com rating, caso ele tenha sido classificado e decrementar a *ratedShowsCounter*.

A seguir será necessário remover o programa do dicionário que contêm os programas com certa *tag*. Caso aquele seja o último programa com a tag, a tag é removida da estrutura que contêm todas as tags.

Por fim, remover o programa da estrutura que contêm todos os programas.

|  |  |  |  |
| --- | --- | --- | --- |
| Operação | Melhor Caso | Pior Caso | Caso Esperado |
| Pesquisa do programa | O(1) | O(*n1*) | O(1) |
| Remover programa de cada pessoa | O(1) | O(*n2* \* log(*n3*)) = O(*x*) | O(*n2* \* log(*n3*)) = O(*x*) |
| Remover programa do vetor de classificações | O(1) | O(log(*n4*)) = O(*y*) | O(log(*n4*)) = O(*y*) |
| Remover programa da estrutura das *tags* | O(1) | O(*n5* \* log(*n6*)) = O(*z*) | O(*n5* \* log(*n6*)) = O(*z*) |
| Remover tag se não houver programas com ela | O(1) | O(*n7*) | O(1) |
| Remover programa da estrutura com todos os programas | O(1) | O(*n1*) | O(1) |
| Complexidade total | O(1) | O(*max(n1, n7, x, y, z)*) | O(*max(x, y, z)*) |

Legenda

*n1* – Número total de programas;

*n2* – Número de pessoas num programa;

*n3* – Número de programas numa pessoa;

*n4* – Número de programas com o mesmo rating do programa a remover;

*n5* – Número de *tags* que um programa tem;

*n6* – Número de programas com certa *tag*;

*n7* – Número total de *tags*.

### Categorizar programa

É atribuída uma *tag* a um dado programa, então precisamos de procurar o programa no dicionário *showsByID* e adicioná-la à sua lista de *tags*.

Depois, também teremos de a procurar no dicionário *listOfShowsByTag*, e adicionar o programa à lista ordenada associada a essa *tag* (caso ela esteja nula, cria-se uma nova lista), que estará ordenada pelo título do programa.

|  |  |  |  |
| --- | --- | --- | --- |
| Operação | Melhor Caso | Pior Caso | Caso Esperado |
| Procura do programa | O(1) | O(*n1*) | O(1) |
| Inserção da *tag* na lista do programa | O(1) | O(1) | O(1) |
| Procura da lista de programas com aquela *tag* | O(1) | O(*n2*) | O(1) |
| Inserção do programa na lista ordenada | O(1) | O(log(*n3*)) | O(log(*n3*)) |
| Complexidade total | O(1) | O(*max(n1, n2*))\* | O(log(*n3*)) |

\* *n1 > n3*

Legenda

*n1* – Número total de programas;

*n2* – Número de *tags*;

*n3* – Número de programas que possuem aquela *tag*.

### Consultar dados de programa

São observados os dados de um certo programa, sendo apenas necessário procurá-lo no dicionário *showsByID*, verificando se ele existe, e listar a sua informação e as suas *tags,* pelaordem a que foram associadas aquele programa.

|  |  |  |  |
| --- | --- | --- | --- |
| Operação | Melhor Caso | Pior Caso | Caso Esperado |
| Procura do programa | O(1) | O(n1) | O(1) |
| Criação do iterador de *tags* | O(1) | O(1) | O(1) |
| Iteração das *tags* do programa | O(*n2*) | O(*n2*) | O(*n2*) |
| Complexidade total | O(*n2*) | O(max(*n1, n2)*) | O(*n2*) |

Legenda

*n1* – Número total de programas;

*n2* – Número de *tags* associadas aquele programa.

### Avaliar programa

É adicionado uma classificação ao programa. Para isso é necessário pesquisar o programa.

Após isso calcular a nova classificação e por fim inserir ou alterar a posição do programa no vetor de ratings, sendo que os programas ficam ordenados por título. Caso o programa não tivesse nenhuma classificação, a variável auxiliar *ratedShowsCounter* é incrementada.

|  |  |  |  |
| --- | --- | --- | --- |
| Operação | Melhor Caso | Pior Caso | Caso Esperado |
| Procura do programa | O(1) | O(*n1*) | O(1) |
| Calcular nova classificação | O(1) | O(1) | O(1) |
| Inserção ou alteração de posição no vetor de ratings | O(1) | O(log(*n2*)) | O(log(*n2*)) |
| Complexidade total | O(1) | O(*max(n1,* log(*n2*)) | O(log(*n2*)) |

Legenda

*n1* – Número total de programas;

*n2* – Número de programas com a mesma classificação.

### Consultar dados de profissional

É listada a informação de um certo profissional, sendo apenas necessário procurá-lo no dicionário *personByID*, verificando se ele existe.

|  |  |  |  |
| --- | --- | --- | --- |
| Operação | Melhor Caso | Pior Caso | Caso Esperado |
| Procura da pessoa | O(1) | O(*n1*) | O(1) |
| Complexidade total | O(1) | O(*n1*) | O(1) |

Legenda

*n1* – Número total de pessoas

### Listar programas associados a um profissional

São listados todos os programas em que um dado profissional participou, por ordem de *ID*, então, procura-se o profissional no dicionário *personByID*, onde se verifica se ele existe e se tem participações, depois é iterada a lista ordenada de programas associados a essa pessoa.

A complexidade de encontrar o próximo elemento a iterar varia entre 1 e *log(n),* sendo o *n* a altura da árvore. Mas esse *log(n)* pode ser desprezado uma vez que a iteração é linear.

|  |  |  |  |
| --- | --- | --- | --- |
| Operação | Melhor Caso | Pior Caso | Caso Esperado |
| Procura da pessoa | O(1) | O(*n1*) | O(1) |
| Criar iterador de programas | O(log(*n2*)) | O(log(*n2*)) | O(log(*n2*)) |
| Iteração de programas | O(*n2*) | O(*n2*) | O(*n2*) |
| Complexidade total | O(*n2*) | O(max(*n1,* *n2)*) | O(*n2*) |

Legenda

*n1* – Número de pessoas

*n2* – Número de programas onde aquele profissional participou.

### Listar participações em programas

São listadas as participações feitas num dado programa, pela ordem que foram feitas, logo será necessário pesquisar no dicionário *showsByID*, verificando se o programa existe e tem alguma participação, e depois iterar pelas participações desse programa

|  |  |  |  |
| --- | --- | --- | --- |
| Operação | Melhor Caso | Pior Caso | Caso Esperado |
| Procura do programa | O(1) | O(*n1*) | O(1) |
| Criar iterador de participações | O(1) | O(1) | O(1) |
| Iteração das participações | O(*n2*) | O(*n2*) | O(*n2*) |
| Complexidade total | O(*n2*) | O(*max*(*n1*, *n2*) | O(*n2*) |

Legenda

*n1* – Número de programas

*n2* – Número de participações feitas naquele programa.

### Listar os melhores programas

É/São listado o(s) programa(s) com a classificação mais alta, por ordem de *ID*, então verificamos se existem programas já fora de produção com alguma classificação, e depois descobrimos qual delas é a mais alta, e vamos a essa posição no vetor de listas ordenadas *listOfShowsByRating*, iterando os programas dessa lista.

A complexidade de encontrar o próximo elemento a iterar varia entre 1 e *log(n),* sendo o *n* a altura da árvore. Mas esse *log(n)* pode ser desprezado uma vez que a iteração é linear.

|  |  |  |  |
| --- | --- | --- | --- |
| Operação | Melhor Caso | Pior Caso | Caso Esperado |
| Descobrir rating mais alta | O(1) | O(1) | O(1) |
| Criar iterador de programas | O(log(*n1*)) | O(log(*n1*)) | O(log(*n1*)) |
| Iteração dos programas | O(*n1*) | O(*n1*) | O(*n1*) |
| Complexidade total | O(*n1*) | O(*n1*) | O(*n1*) |

Legenda

*n1* – Número de programas com a classificação mais alta.

### Listar programas com avaliação

É/São listado o(s) programa(s) com a classificação dada, por ordem alfabética do título. Como no método anterior, verificamos se existem programas já fora de produção com a classificação dada, e caso existam, vamos à posição do vetor que corresponde ao número dessa classificação, e iteramos os programas dessa lista.

A complexidade de encontrar o próximo elemento a iterar varia entre 1 e *log(n),* sendo o *n* a altura da árvore. Mas esse *log(n)* pode ser desprezado uma vez que a iteração é linear.

|  |  |  |  |
| --- | --- | --- | --- |
| Operação | Melhor Caso | Pior Caso | Caso Esperado |
| Criar iterador de programas | O(log(*n1*)) | O(log(*n1*)) | O(log(*n1*)) |
| Iteração dos programas | O(*n1*) | O(*n1*) | O(*n1*) |
| Complexidade total | O(*n1*) | O(*n1*) | O(*n1*) |

Legenda

*n1* – Número de programas com a classificação dada.

### Listar programas com palavra-chave (*tag*)

É/São listado o(s) programa(s) com a *tag* dada, por ordem alfabética do título, logo verifica-se se existem programas com a *tag* dada, e, caso existam, procura-se no dicionário *listOfShowsByTag* pela lista ordenada de programas, e caso exista, iteramo-los.

A complexidade de encontrar o próximo elemento a iterar varia entre 1 e *log(n),* sendo o *n* a altura da árvore. Mas esse *log(n)* pode ser desprezado uma vez que a iteração é linear.

|  |  |  |  |
| --- | --- | --- | --- |
| Operação | Melhor Caso | Pior Caso | Caso Esperado |
| Pesquisar pela lista de programas | O(1) | O(*n1*) | O(1) |
| Criar iterador de programas | O(log(*n2*)) | O(log(*n2*)) | O(log(*n2*)) |
| Iteração dos programas | O(*n2*) | O(*n2*) | O(*n2*) |
| Complexidade total | O(*n2*) | O(max(n1, *n2)*) | O(*n2*) |

Legenda

*n1* – Número de *tags*

*n2* – Número de programas com a *tag* dada.

### Terminar execução

A aplicação para de aceitar comandos novos e guarda a informação que recebeu até agora num ficheiro.

|  |  |  |  |
| --- | --- | --- | --- |
| Operação | Melhor Caso | Pior Caso | Caso Esperado |
| Complexidade total | O(1) | O(1) | O(1) |

# Complexidade espacial

A complexidade espacial do nosso programa será a soma do tamanho ocupado por todas as estruturas de dados.

As estruturas de dados e os dados que as mesmas contêm estão descritos no capítulo 2.2 deste relatório.

A complexidade é dada pela seguinte formula:

O(*d1*) + O(*d2*) + O(11 \* *d3*) + O(*d4* \* *d5*) + O(*np* \* *d6*) + O(*nP* \* (*d7* + *d8* + *d9*))

Legenda

*d1* – Tamanho *SepChainHashTable* que guarda os programas;

*d2* – Tamanho *SepChainHashTable* que guarda as pessoas;

*d3* – Tamanho *AVLTreeE* que guardar os programas classificados com certo *rating*;

*d4* – Tamanho *SepChainHashTable* que guarda uma *AVLTreeE*, com tamanho *d5*;

*d5* – Tamanho *AVLTreeE* que guarda programas com certa *tag*;

*np* – Número de pessoas;

*d6* – Tamanho *AVLTree* que guarda os programas onde certa pessoa participa;

*nP* – Número de programas;

*d7* – Tamanho da *DoubleList* que guarda as participações de um programa;

*d8* – Tamanho da *DoubleList* que guarda as *tags* de um programa;

*d9* – Tamanho da *SepChainHashTable* que guarda as pessoas que participaram num programa.

# Conclusão

Durante o desenvolvimento deste projeto, nomeadamente no desenvolvimento das estruturas de dados, conseguimos compreender essas estruturas de dados funcionam e quais as vantagens e desvantagens de usar certa estrutura.

Foi um projeto que nos fez pensar e discutir sobre quais seriam as formas mais eficientes de fazer as várias funções de forma que o programa fosse o mais eficiente possível.