

# Projeto de Laboratórios de Informática 3

## Grupo 21

Diogo Braga A82547      João Silva A82005

Ricardo Caçador A81064

12 de Junho de 2018

### Resumo

Este documento apresenta o projeto de Laboratórios de Informática 3 (LI3), do curso de Engenharia Informática da Universidade do Minho.

O projeto baseia-se na criação de um sistema de análise de ficheiros XML que possuem informações do Stack Overflow, um website de perguntas e respostas sobre programação de computadores.

Agora realizado em Java, este projeto tem por base um anterior realizado em C, de modo que o relatório possui muitas comparações de resoluções e questões de eficiência entre as duas linguagens.

## Conteúdo

<b>1</b>	<b>Introdução</b>	<b>2</b>
<b>2</b>	<b>Estruturas de Dados</b>	<b>2</b>
<b>3</b>	<b>Modularidade</b>	<b>2</b>
<b>4</b>	<b>Encapsulamento</b>	<b>3</b>
<b>5</b>	<b>Abstração de Dados</b>	<b>3</b>
<b>6</b>	<b>Estratégias das Queries</b>	<b>3</b>
6.1	Init . . . . .	3
6.2	Load . . . . .	3
6.3	Query 1 . . . . .	4
6.4	Query 2 . . . . .	4
6.5	Query 3 . . . . .	4
6.6	Query 4 . . . . .	4
6.7	Query 5 . . . . .	5
6.8	Query 6 . . . . .	5
6.9	Query 7 . . . . .	5
6.10	Query 8 . . . . .	6
6.11	Query 9 . . . . .	6
6.12	Query 10 . . . . .	6
6.13	Query 11 . . . . .	7
6.14	Clean . . . . .	7

7	Conclusão	7
8	Bibliografia	7

## 1 Introdução

Este trabalho tem por base o parse de elementos de ficheiros XML relacionados com a informação do website Stack Overflow, de forma a seguidamente responder a uma série de questões relacionadas com posts, utilizadores e tags do mesmo website. Aliado a tal está o desafio de procurar sempre o melhor algoritmo de resolução das queries, de forma a tornar o código mais eficiente e o mais rápido possível. Em todas as secções do relatório existe uma área dedicada à comparação das duas formas de resolução, em C e Java respetivamente.

A Secção 2 apresenta as estruturas de dados utilizadas no projeto, a Secção 3 aborda a modularidade, a Secção 4 aborda o encapsulamento, a Secção 5 aborda a abstração de dados e a Secção 6 indica as estratégias usadas para resolver as questões apresentadas. O relatório termina com conclusões na Secção 7, onde é também apresentada uma análise crítica dos resultados obtidos.

## 2 Estruturas de Dados

Este trabalho tem por base uma classe principal denominada **TCD\_community**. Esta estrutura possui:

- Classe **Utilizador** que contem um **Map** em que as Keys são os **Id**'s dos Utilizadores e os Values são as instâncias da classe **Utilizador**:
  - Map <Long,Utilizador>**utilizador**
- Classe **Posts** que contem um **Map** em que as Keys são os **Id**'s dos Posts e os Values são as instâncias da classe **Posts**:
  - Map <Long,Posts>**posts**
- Classe **Tag** que contem um **Map** em que as Keys são os **Name**'s das Tags e os Values são as instâncias da classe **Tag**:
  - Map <String,Tag>**tag**

No projeto anterior tivemos a necessidade de ter uma estrutura com os posts ordenados por data. Em Java fazemos uso da interface **Comparator** implementada na classe requerida, e em run-time, usando as propriedades dos **TreeSet**'s, organizamos os posts úteis para a resolução da query.

## 3 Modularidade

Modularidade, por definição, é a divisão do código fonte em unidades separadas coerentes. Esta torna-se fundamental para lidar com a complexidade do código, de tal forma que o código dos programas deve ser dividido por unidades modulares pequenas e autónomas.

Modularidade é algo que temos em conta no nosso projeto em Java, desta vez através da realização do MVC, constituída por:

- O **Model** define a camada computacional que contém os dados e os algoritmos para o seu processamento.
- A **View** define a camada de apresentação e interação com o utilizador.
- O **Controller** define a camada fundamental de controlo do fluxo de execução da aplicação, é o mediador entre o model e a view.

## 4 Encapsulamento

Encapsulamento baseia-se na garantia de **protecção e acessos controlados aos dados**. É mais um aspeto que temos em conta no nosso projeto de forma a que exista uma divisão entre as operações que são públicas e aquelas que são internas ao módulo. Estas são privadas e, portanto, são apenas acessíveis do exterior através das funções disponibilizadas na **API**.

No projeto de C o encapsulamento não era total, isto é, era utilizada por base uma abordagem *shallow*. Em Java, o encapsulamento é total de modo que é realizado *deep clone*. Notar que o uso da primeira abordagem torna o programa muito mais rápido devido à não necessidade de clonar totalmente os objetos. Por outro lado, usando esta abordagem em Java, temos a vantagem de que não existe partilha de apontadores.

## 5 Abstração de Dados

A declaração abstrata duma estrutura esconde dos utilizadores do módulo a implementação concreta, não tendo desta forma acesso à implementação da mesma. Por isso mesmo, previamente, temos a declaração abstrata da classe **TCD\_Community**, denominada de **TAD\_Community**, usada como interface para as queries.

## 6 Estratégias das Queries

### 6.1 Init

Função que cria a **TCD\_community**.

### 6.2 Load

Método que realiza o parser em **chunks** dos ficheiros necessários à realização do trabalho, através da API do StAX. Após receber um **dump\_path** para os ficheiros XML, caso não existam falhas na estrutura XML são realizadas três funções: o **getReferenceUser**, o **getReferencePosts** e o **getReferenceTags**.

A primeira realiza todo o parse relacionado com os **utilizadores**, como por exemplo o **id** ou a **reputação**. De seguida são colocados na estrutura todos estes elementos através do método **set\_utilizador**.

A segunda realiza todo o parse relacionado com os **posts**, como por exemplo o **id\_post** ou o **post\_type\_id**. De seguida são colocados na estrutura todos estes elementos através do método **set\_posts**.

A terceira realiza todo o parse relacionado com as **tags**, como por exemplo o **id\_tag** ou o **tag\_name**. De seguida são colocados na estrutura todos estes elementos através do método **set\_tag**.

### 6.3 Query 1

Dado o identificador de um post, a função retorna o título do post e o nome de utilizador do autor. Se o post for uma resposta, a função retorna o título e o id do utilizador da pergunta correspondente.

Nesta questão, criamos um **Map** onde colocamos todos os posts da comunidade. Se o ID que recebemos como parâmetro pertencer ao **Map** podemos prosseguir com a realização da query.

Criamos um novo map com todos os utilizadores da comunidade. Caso o post seja uma pergunta, retornamos o par com o título do post e o nome do user.

Caso contrário, retornamos o par com o título e o ID da pergunta (**parent\_id**).

### 6.4 Query 2

Pretendemos obter o top N utilizadores com maior número de posts de sempre. Para isto, são considerados tanto perguntas quanto respostas dadas pelo respectivo utilizador.

Nesta questão, criamos um **TreeSet** de clones de **Utilizador(es)**, com a intenção de os ordenar decrescentemente tendo em conta o **número de posts** realizados por cada um. Tal é realizado pelo **ComparatorPosts** que implementa a classe Utilizador.

Seguidamente, através dum **Iterator**, todos os objetos recebem o módulo **get\_key\_id** para retornar os **N Id's** passados como parâmetro na query.

### 6.5 Query 3

Dado um intervalo de tempo arbitrário, obtemos o número total de posts (identificando perguntas e respostas separadamente) neste período.

Nesta questão, inicializamos **2 variáveis que vão contar o número de perguntas e o número de respostas** que estão dentro da data recebida. Assim, percorremos todos os posts e verificamos se é pergunta ou resposta, incrementando a respetiva variável.

Por fim, basta colocar na estrutura **Pair**, as respetivas variáveis.

### 6.6 Query 4

Dado um intervalo de tempo arbitrário, retornamos todas as perguntas contendo uma determinada tag. O retorno da função é uma lista com os IDs das perguntas ordenadas em cronologia inversa.

Nesta questão, criamos um **TreeSet** que implementa o **Comparator** de Datas com a intenção de retornar uma lista de ID's ordenada cronologicamente.

Ou seja, através de um ciclo for, percorremos todos os posts da comunidade e, para cada um, após verificar que é um **Post** do tipo pergunta, verificamos também se este se encontra dentro das datas recebidas. Caso verifique ambas

as condições é criada uma **lista de strings** na qual são colocadas as **tags** do respectivo post.

Finalizando, verificamos se essa lista de tags contém a tag que queremos verificar se existe na pergunta e caso se confirme, o post é clonado e adicionado ao **TreeSet**. Como queremos retornar os ID's das perguntas criamos uma lista de Long's e um **Iterator** que percorre o **TreeSet** e coloca os ID's na lista a ser retornada como resultado.

## 6.7 Query 5

Dado um ID de utilizador, devolvemos a informação do seu perfil (short bio) e os IDs dos seus 10 últimos posts (perguntas ou respostas), ordenados por cronologia inversa.

Nesta questão, criamos um **TreeSet** que implementa o **ComparadorData** que permite inserir os **Posts** no Set ordenados cronologicamente.

Assim, o primeiro passo é percorrer todos os posts da comunidade e colocar no **TreeSet** todos os posts cujo **owner\_user\_id** seja o do utilizador que recebemos.

Depois, criamos um **ArrayList** e um **Iterator** que vai percorrer o **TreeSet** e adicionar os ID's à lista até um total de 10 ID's de utilizadores. Por fim, retornamos o Pair com a bio do utilizador e a lista dos ID's dos seus últimos 10 posts.

## 6.8 Query 6

Dado um intervalo de tempo arbitrário, devolver os IDs das N respostas com mais votos, em ordem decrescente do número de votos.

De referir que tivemos em conta o score das perguntas, ao invés da diferença de votos.

Nesta questão, recorremos mais uma vez à implementação de um Comparador, neste caso, o **ComparadorScore** que vai permitir, na função, colocar no **TreeSet** os posts em ordem decrescente.

Então, depois de criar o **TreeSet** que implementa este Comparador, percorremos todos os posts da comunidade e, depois de verificar se é um post do tipo resposta e se está entre as datas recebidas como parâmetros, colocamos no **TreeSet**.

Por fim, utilizando o mesmo método de queries anteriores, criamos uma lista que vai conter os ID's e um iterator para percorrer o Set com os posts.

Retornamos assim, uma lista com os IDs das N respostas com mais votos.

## 6.9 Query 7

Dado um intervalo de tempo arbitrário, devolver os IDs das N perguntas com mais respostas, em ordem decrescente do número de votos.

De referir que as respostas tidas em conta abordam o tempo total, e não o intervalo de tempo passado como parâmetro.

Nesta questão, implementamos o **ComparadorAnswer**, para permitir a organização decrescente das perguntas tendo em conta o número de respostas.

Depois de criarmos o `TreeSet` que implementa este `Comparator`, percorremos todos os posts, verificando se é uma pergunta e se está dentro das datas dos parâmetros e, caso se verifique, a pergunta é adicionada ao `TreeSet`.

Pelo mesmo método de queries anteriores, criamos um `ArrayList` de `Longs` que vai conter os IDs dessas perguntas, que irão ser adicionados através de um `while`, implementando um `Iterator`.

### 6.10 Query 8

Dado uma palavra, devolver uma lista com os IDs de N perguntas cujos títulos a contenham, ordenados por cronologia inversa.

Nesta questão, criamos um `TreeSet` que implementa o **`ComparatorData`** (já utilizado anteriormente) que permite a ordenação cronológica dos posts num `TreeSet`.

Percorremos todos os posts da comunidade e, após verificar que é uma pergunta, verificamos se o título desse post contém a `String` recebida nos parâmetros. Caso isso se verifique o post é clonado e adicionado ao `TreeSet` auxiliar.

Depois, através do método já usado em outras queries, com um `Iterator`, percorremos esse `TreeSet` adicionando a uma `List` de `Longs` os ID's dos posts.

### 6.11 Query 9

Dados os IDs de dois utilizadores, devolver as últimas N perguntas, em cronologia inversa, em que participaram dois utilizadores específicos, via pergunta ou respostas.

Nesta questão recorremos à criação de um **`Map`** com todos os utilizadores. De seguida, verificamos se o ID de ambos os utilizadores pertence a esse `Map`. Caso se verifique, colocamos numa lista (para cada utilizador) os IDs dos posts nos quais estes utilizadores interagiram.

Assim, basta depois percorrer a primeira lista e comparar com os IDs da segunda lista através do método **`contains()`** e adicionar a um **`TreeSet`** que implementa o **`ComparatorData`** para os posts estarem por ordem cronológica.

Por fim, como em queries anteriores e porque queremos apenas os IDs dos posts, criamos um `Iterator` para percorrer o `TreeSet` e adicionar a uma `List` de **`Longs`**.

### 6.12 Query 10

Dado o ID de uma pergunta, obter a melhor resposta, tendo em conta uma média ponderada.

Nesta questão, utilizamos uma **`Map <Long,Posts>`** que contém todos os posts da comunidade.

Primeiro, verificamos se o ID da pergunta que recebemos como parâmetro pertence ao `Map` com todos os posts e se é uma pergunta (pois queremos analisar as respostas dessa pergunta)

Passando estas verificações, criamos outro `Map`, desta feita com todos os utilizadores da comunidade. Depois, percorremos todos os posts da comunidade, e sempre que for um **`Post.resposta`**, verificamos se o **`parent_id`** da resposta é o ID que recebemos como parâmetro. Caso se confirme, calculamos a média através da equação fornecida e se esta for melhor que a melhor média calculada

até aquela resposta, esta é substituída e colocamos no **melhor\_id** o ID da resposta.

### 6.13 Query 11

Dado um intervalo arbitrário de tempo, devolver os identificadores das N tags mais usadas pelos N utilizadores com melhor reputação. Em ordem decrescente do número de vezes em que a tag foi usada.

Nesta questão, implementamos 2 Comparador's, um para compara as ocorrências de cada tag (ComparatorOcorrencias) e outro que compara a reputação dos utilizadores (ComparatorReputacao).

O primeiro passo é a criação das estruturas. Criamos um **Map** que tem todos os utilizadores e um **TreeSet** de utilizadores que implementa o ComparatorReputacao.

Assim no primeiro ciclo for, adicionamos ao **TreeSet** os utilizadores para ficarem organizado por reputação.

Depois, criamos um **Iterator** que vai percorrer o **TreeSet** dos utilizadores ordenados por reputação e, para utilizador, vai percorrer os seus posts\_perguntas, verificando se estes se encontram dentro dos parâmetros das datas. Caso se verifique, percorremos todas as tags desse post e para cada tag verificamos se pertence à estrutura todas.tags (contém todas as tags) e incrementamos uma variável que conta as ocorrências, se essa tag pertencer. Caso contrário, adicionamos à estrutura todas.tags essa nova tag.

Após analisarmos as tags dos posts, criamos um TreeSet de **TagUnique** que implementa o **ComparatorOcorrencias** para colocar no TreeSet, de forma ordenada por nº de ocorrências cada uma das tags.

Por fim, e como queremos retornar os identificadores das N tags mais usadas, criamos uma List de Long e, através de um Iterator, colocamos nessa List, N ID's de tags.

### 6.14 Clean

Função que limpa toda a estrutura da comunidade com o auxílio do **GarbageCollection**.

## 7 Conclusão

////////// FALTA FAZER //////////

Tendo em conta os objetivos definidos, o grupo acha que os resultados principais foram obtidos.

Depois da realização deste mesmo projeto podemos declarar várias diferenças entre os dois projetos, denominadamente:

- 

## 8 Bibliografia

1. Prof. F. Mário Martins, *Programação Modular em C*;

2. Prof. F. Mário Martins, *Tipos Incompletos em C*.
3. Prof. F. Mário Martins, *Modelo MVC*.