Universidade do Minho

Processamento de xml

Mestrado Integrado em Engenharia Informática

Laboratórios de Informática 3

(2º Ano, 2º Semestre, 2017/2018)

A79003 Pedro Mendes Félix da Costa

A80453 Bárbara Andreia Cardoso Ferreira

Braga
Junho 2018

Índice

1. Introdução

Este trabalho foi feito no âmbito da unidade curricular laboratórios de informática 3 e tem como objetivo o processamento de qualquer *dump* da base de dados do site StackOverflow para responder a queries de forma eficiente, aplicando conhecimentos de algoritemia e programação orientada a objectos.

2. Resolução inicial do Problema

Fazendo uma análise às queries, decidimos que seria necessário representar as seguintes entidades:

2.1 Posts

Para representar um **Post** criamos a classe abstrata do mesmo nome. Esta guarda os dados e representa o comportamento comum a todos os posts. Depois, para cada tipo de post considerado, questão ou resposta, foi criada uma classe que extende **Post**, adicionando os dados únicos deste tipo.

Post

- Id
- Score
- Data
- Id do autor
- Nome do autor

Questão

- Título da questão
- Número de respostas
- Lista das respostas
- Tags

Resposta

- Número de comentários
- Id da questão a que responde
- Questão a que responde

Além dos dados fornecidos diretamente pelos ficheiros xml, decidimos também guardar na **questão** a lista das **respostas** de cada questão. Na **resposta**, inversamente, guardamos a questão a que esta responde. Com estas informações extra, as pesquisas que envolvem relacionar estas duas entidades tornam-se muito mais eficientes.

2.2 Utilizadores

Para representar os **utilizadores** guardamos os seguintes atributos:

- Id
- Biografia
- Nome
- Reputação
- Número de posts
- Lista dos posts

Mais uma vez foram guardadas mais informações para além das disponibilizadas diretamente pelo xml. Foi guardado o número de **posts** do utilizador (para determinar os utilizadores mais ativos de forma mais rápida) e a lista destes para permitir pesquisas mais rápidas.

Esta lista de posts faz uso de uma coleção concebida para resolver este tipo de problemas. Neste caso em particular, uma lista ligada ordenada (secção 3.4) por data. Devido à forma quase ordenada em que os posts surgem no xml esta ordenação é facilmente mantida.

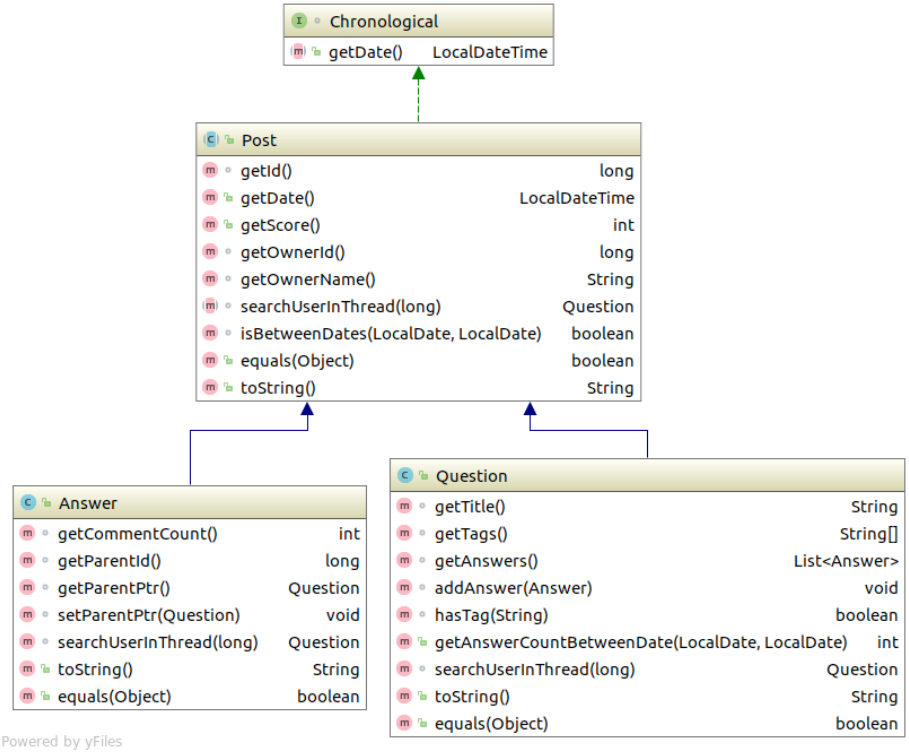

Figura 2.1: Hierarquia formada pelos posts

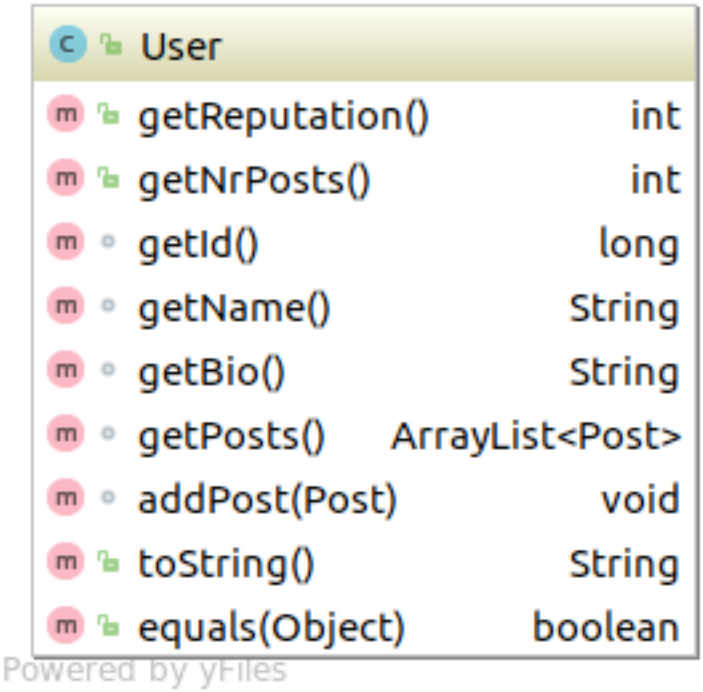

Figura 2.2: Classe do utilizador

3. Estruturas de Dados

3.1 Maps

Todas as entidades são armazenadas numa tabela de hash pois para todas são necessárias pesquisas por id (ou nome no caso das tags).

3.1.1 Tags

A tabela de hash das *tags* serve para criar uma associação $Nome \rightarrow Id$ visto que as questões guardam uma lista com os nomes das tags e para responder à query 11 é necessário obter os ids das mesmas.

3.2 Calendário

Para ser possível manter as questões e respostas ordenadas por data foi concebida uma estrutura à qual demos o nome de **Calendário** que permite acessos em tempo constante $O(1)$ a todos os elementos associados a uma determinada data. Mantemos assim duas instâncias desta estrutura na **Community**, uma para perguntas e outra para respostas.

Para que esta estrutura fosse genérica criamos também uma interface **Chronological** que tem de ser implementada pelos elementos que são adicionandos à estrutura. Esta interface obriga à implementação do método `LocalDateTime getDate()` para que estes possam ser guardados de forma ordenada.

Assim, esta estrutura permite:

- Guardar qualquer objeto desde que implemente a interface referida anteriormente.
- Iterar sobre os seus elementos, dado um intervalo de tempo, por ordem cronológica normal ou inversa, conforme a ordem dos argumentos.

Para utilizar os métodos de iteração, é necessário, para além do intervalo de tempo, passar um *Predicate* que é uma classe que implementa a interface funcional do mesmo nome. Esta, define um método que será aplicado a todos os elementos do **Calendario** por ordem cronológica, inversa ou não, dependendo da ordem em que as datas são passadas para definir o intervalo.

A estrutura em si consiste num `Map` que associa a cada data uma lista ordenada dos elementos desse data.

A utilização de listas ligadas para guardar os elementos de um dia é mais eficiente, pois é necessário fazer inserções de forma ordenada, que são, em quase todos os casos, feitas à cabeça devido aos posts serem inseridos quase cronologicamente. Caso não ouvesse esta certeza, estas poderiam ser substituídas por um `TreeSet` para que as inserções fossem no pior caso $O(log(N))$.

3.3 Community

Para armazenar as entidades descritas acima foi implementada uma classe chamada **Community** que as armazena de forma a mais tarde facilitar as queries que terão de ser feitas sobre estas.

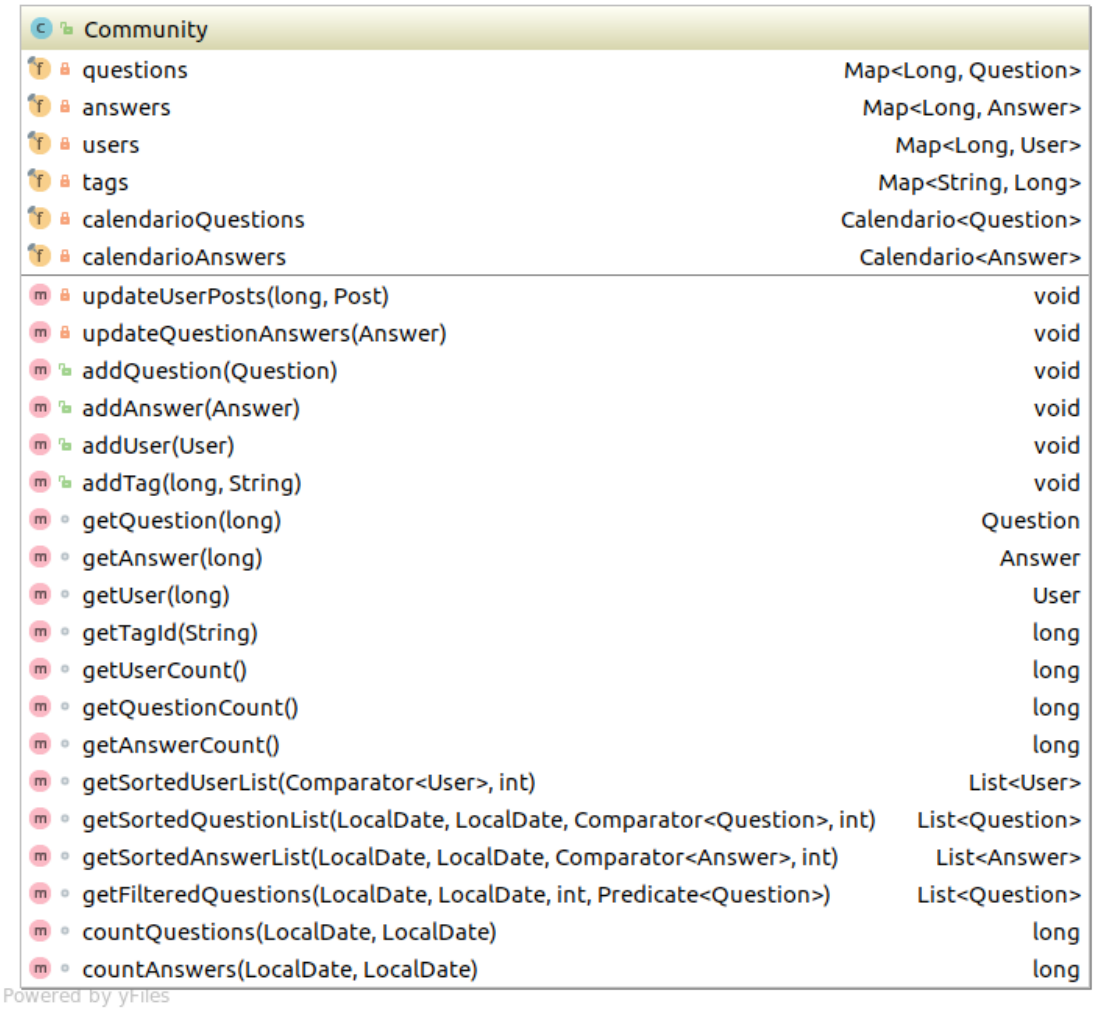

Figura 3.1: Community

3.4 SortedLinkedList

Foi necessário desenvolver um novo tipo de lista para satisfazer as necessidades de algumas estruturas. Esta lista extende `LinkedList` adicionando a noção de ordem. Para que seja eficiente, esta deve ser usada quando a maior parte das inserções forem feitas à cabeça ou na cauda da lista. Opcionalmente, esta pode também ter um tamanho máximo.

3.5 TCD

A forma como o nosso modelo interage com o exterior é atravez da classe `TCD`. Esta implementa todas as queries necessãrias de forma simples devido à maior parte do processamento ser feito pelos metodos disponiveis no `Community`.

3.6 Arquitetura

Todas estas classes formam a nossa arquitetura que pode ser vista de forma esquematizada na Figura 3.2.

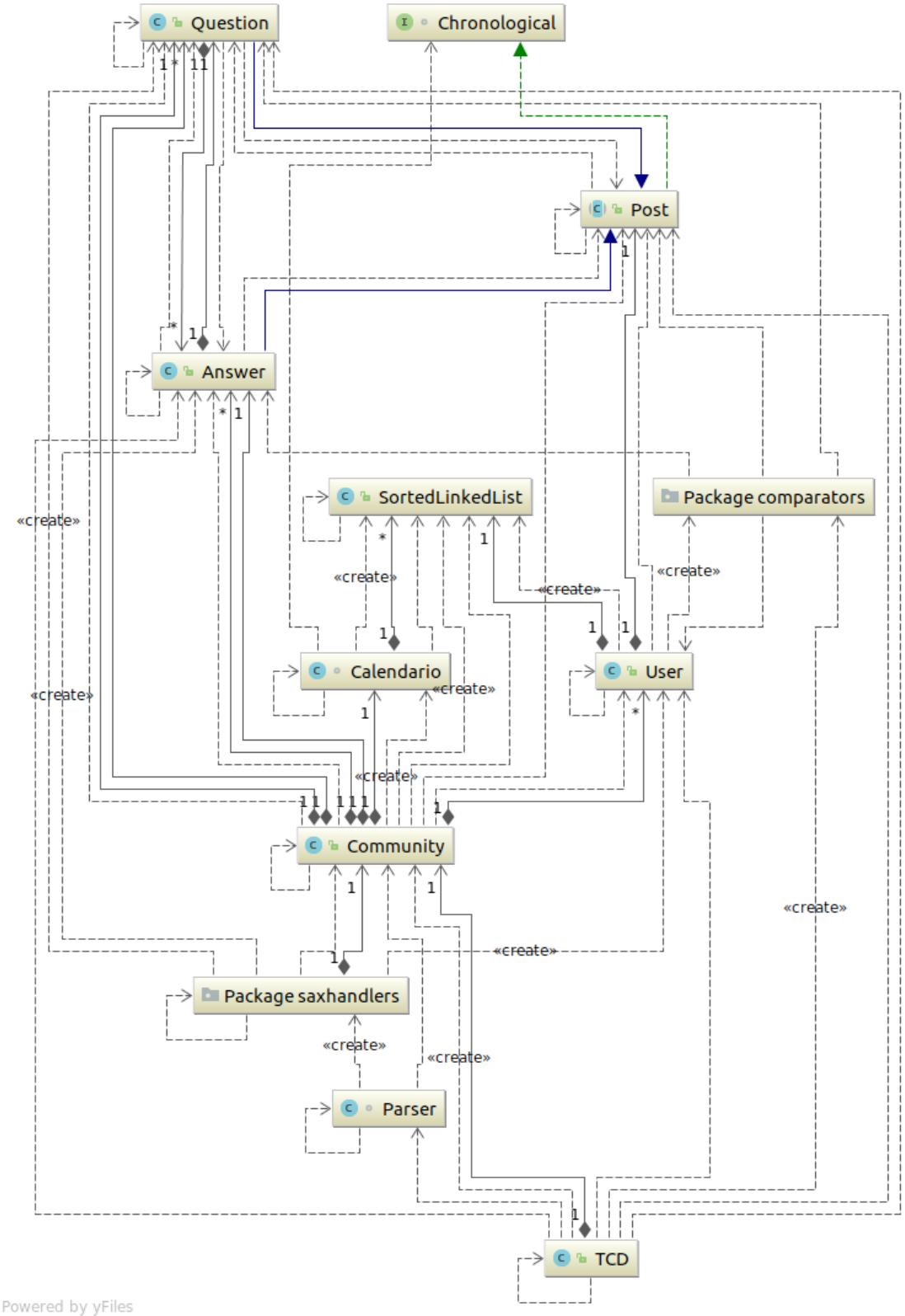

Figura 3.2: Diagrama de Classes do Engine

4. Modelo MVC e Encapsulamento

A arquitetura de packages desenvolvida é a seguinte:

- **common** Onde estão definidas as classes utilitárias.
- **engine** Onde está definido o modelo.
- **view** Onde está definida a view.
- **li3** Onde está definido o controlador.

Devido ao uso de packages todas as classes que sejam públicas para fora da package, não têm métodos públicos que permitam alterar o seu estado interno. Desta forma, todos os objetos aparecem imutáveis para o exterior.

A **view** define os métodos para interagir com o utilzador.

O **engine** ou *Modelo* define as operações que se podem ser realizadas sobre os dados.

A package **li3** tem o controlador que faz a comunicação entre os pedidos do utilizador e as operações do modelo.

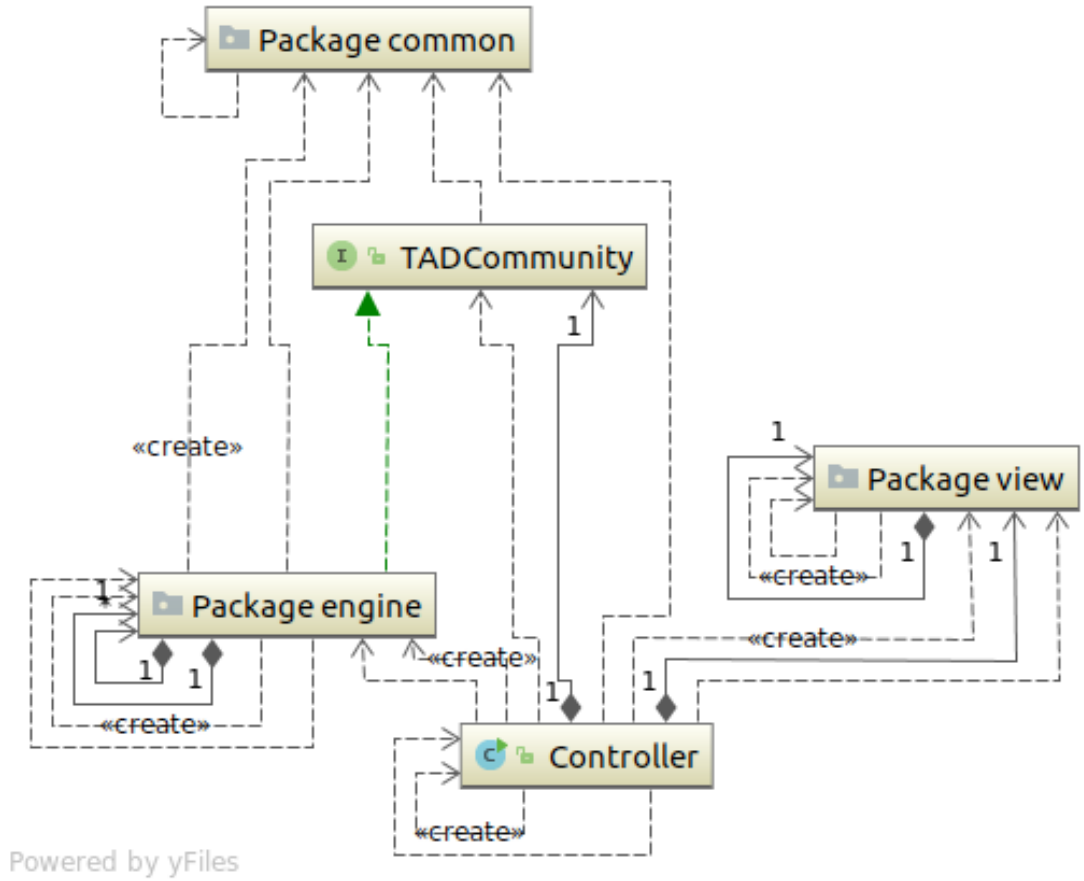

Figura 4.1: Diagrama MVC

5. Modularização Funcional e Resolução das queries

Para aceder aos dados da estrutura principal foi definida uma API simples que permite:

1. Pesquisas por id de **questões**, **respostas** e **utilizadores**.
2. Pesquisas de ids de **tags** dada a designação.
3. Pesquisa de listas, ordenadas por qualquer critério, de **utilizadores**, **questões** e **respostas**.
4. Pesquisas de **questões** filtradas por qualquer critério.
5. Contagens de **questões** ou **respostas** num determinado intervalo de tempo.

Com estas funções a resolução da maioria das queries mostrou-se trivial.

Para as queries que necessitam de pesquisas por id (queries: 1, 5, 9 e 10) são resolvidas por **1**.

Para as queries que necessitam de pesquisas de utilizadores ordenados (queries: 2 e 11) são conseguidas através de **3**, sendo que assim basta fornecer um comparator para definir o critério de ordenação.

Para as queries que necessitam de pesquisas de **perguntas/respostas** num intervalo de tempo ordenadas (queries: 6 e 7) são também resolvidas através de **3**. No caso de não ser necessária a filtragem do intervalo de tempo, simplesmente passamos as datas máximas disponibilizadas: `LocalDate.MAX` `LocalDate.MIN`.

Para as queries que necessitam de listas de questões que obedeçam a um determinado critério (queries: 4 e 8), são conseguidas através de **4**.

Para obter o numero de **perguntas** e **respostas** dentro de um intervalo de tempo (query: 3) fizemos uso dos metodos desenvolvidos para este mesmo proposito **5**.

A resolução da query 11 tem quatro fases. A primera consiste em obter o top N utilizadores (baseado na reputação destes) para para obter os posts destes. Depois, usando um `Map` conta-se a ocorrência de cada **Tag** nas questões dos utilizadores obtidos. Na terceira fase do problema, colocam-se as tags numa *PriorityQueue* para que a remoção destas seja ordenada por ocorrência, e numa última fase, para cada uma das **Tags** removidas, é adicionado o id desta à lista que será retornada.

6. Comparação com a versão em C

Em termos de eficiência uma linguagem nativa como C é muito mais rápida a executar as queries. Isto notou-se, essencialmente, no parsing.

Tabela 6.1: Comparação de tempos

Query	**C**	**Java**
Load	10912	16827
Query 1	0	0
Query 2	25	103
Query 3	0	1
Query 4	0	5
Query 5	0	1
Query 6	0	6
Query 7	0	9
Query 8	0	3
Query 9	0	1
Query 10	0	0
Query 11	22	102
Total	11922	17058

A estrutura que sofreu mais alterações foi o Calendario, inicialmente transpilamos o codigo para Java, mantendo a matriz de quatro dimemensões. No entanto, quando testamos os tempos das queries verificamos que esta não era de forma alguma eficiente, visto que demorava 8000 ms a calcular o resultado da query 8. Para resolver este problema desenvolvemos uma solução que, no fim, se mostrou relativamente simples.

Em termos de facilidade de debug e desenvolvimento, o código Java é muito mais simples, apesar de não ser uma comparação justa, visto que, tendo o trabalho feito em C de forma estruturada, muitos dos possíveis erros já foram evitados.

7. Conclusões e Trabalho Futuro

Em suma, o grupo considera que o trabalho foi realizado na sua totalidade de forma eficiente e correta, respondendo a todas as queries.

Um aspeto que poderia ser melhorado é a ordenação de utilizadores. Estes foram guardados apenas numa tabela de hash, logo, quando é necessária uma lista ordenada dos mesmos, a tabela tem de ser percorrida na sua totalidade. Esta decisão, centrou-se no facto de que nenhuma única ordenação se apresentar particularmente vantajosa, face às demais.