

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

1	Introdução	2
2	Estruturas de Dados	2
2.1	Comparação com C	2
3	Modularidade	3
4	Encapsulamento	3
5	Abstração de Dados	3
6	Estratégias das Queries	3
6.1	Init	3
6.2	Load	4
6.3	Query 1	4
6.4	Query 2	4
6.5	Query 3	5
6.6	Query 4	5
6.7	Query 5	5
6.8	Query 6	5
6.9	Query 7	6
6.10	Query 8	6
6.11	Query 9	6
6.12	Query 10	7

6.13 Query 11	7
6.14 Clean	7
7 Conclusão	8
8 Bibliografia	8

Lista de Figuras

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**

2.1 Comparação com C

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

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

Modularidade, por definição, é a divisão do código fonte em unidades separadas coerentes. É algo que temos em conta no nosso projeto, existindo, por base, o ficheiro **main.c**, e depois todos os ficheiros individuais relacionados com as **estruturas de dados** usadas e as **queries** propostas. Estes possuem os ficheiros **.c** que contêm o código fonte e os **.h/header files** que definem o que é invocável do exterior.

Modularidade torna-se portanto 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, devendo-se ter em especial atenção a criação dos módulos que representam **abstração de dados**.

4 Encapsulamento

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

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**

Desta forma um tipo de dados permite ter várias instanciações, visto que os módulos das estruturas e das queries se tornam mais genéricos.

De referir que o encapsulamento não é total, isto é, só são feitos clones no caso das strings, através da função **mystrdup**. Ao invés disso utilizamos uma abordagem *shallow*, por uma questão de eficiência tendo sempre o cuidado de não alterar valores das estruturas.

5 Abstração de Dados

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

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 estrutura **utilizador**, denominada de **UTILIZADOR**, no *header file*.

6 Estratégias das Queries

6.1 Init

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

Função que cria a **TCD_community**. Por consequência, inicializa as estruturas relacionadas a essa mesma struct, alocando memória e usando a função da *glib*, **g_hash_table_new**.

6.2 Load

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

Função que realiza o parse de todos os ficheiros necessários à realização do trabalho. 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 da função **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 da função **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 da função **set_tag**.

Referir que incluímos um ficheiro *debug.h* em todas as queries de modo a que seja possível, ou não, imprimir as respostas no standard output.

6.3 Query 1

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

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, sendo o valor do **id** igual ao da **key** da tabela de hash, recorremos à função da *glib*, **g_hash_table_lookup** que dado uma **key**, retorna o **value** associado. Caso nada seja encontrado, é retornado NULL.

Tendo agora todos os valores referentes ao **post**, caso este seja uma pergunta, é atribuído à primeira coordenada o **title** do post e à segunda o **nome** de quem realizou a questão. Encontramos o **nome** do autor da questão invocando o parâmetro **owner_user_id** na mesma função **lookup** utilizada anteriormente, passando agora a ser esse o **key/id** associado.

Caso seja uma resposta, o primeiro parâmetro é calculado usando a mesma função **g_hash_table_lookup**, mas agora com o parâmetro **parent_id**, que numa resposta retorna o **id** da pergunta ao qual esta respondeu. O segundo parâmetro é igualmente calculado como se fosse uma pergunta, mudando apenas o novo **value** associado.

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 respectiva variável.

Por fim, basta colocar na estrutura **Pair**, as respectivas 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 **ComparatorData** 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, recorreremos mais uma vez à implementação de um Comparador, neste caso, o **ComparatorScore** 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 **ComparatorAnswer**, para permitir a organização decrescentes das perguntas tendo em conta o número de respostas.

Depois de criarmos o **TreeSet** que implementa este Comparador, percorremos todos os posts, verificando se é uma pergunta e 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 contêm, 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

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

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 recorreremos à estrutura **utilizador** que contém a informação sobre todos os utilizadores, para retirar a informação relativa aos **posts frequentados**, de cada utilizador que nos são passados como argumentos. Resta apenas comparar estas duas **GList*** para retirar os **id_post** que coincidem.

6.12 Query 10

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

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

Nesta questão, utilizamos uma **GList*** para armazenar os **values** de cada **utilizador**, sendo isto realizado pela função da *glib*, **g_hash_table_get_values**.

Depois, identificamos a pergunta ao qual o **post** está a responder, através do **parent_id**. Caso essa pergunta seja o **id** passado como parâmetro, calculamos a média ponderada tendo em conta o **score**, o **comment_count** e a **reputação**. Este último parâmetro é acedido na estrutura do utilizador através da função **g_hash_table_lookup**. Quando encontramos uma média melhor é alterado o **id_post** a retornar no final.

6.13 Query 11

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

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, utilizamos uma **GList*** denominada **glista** para armazenar os **values** de cada **utilizador**, sendo isto realizado pela função da *glib*, **g_hash_table_get_values**. Esta lista é depois ordenada tendo em conta a reputação através da função **g_list_sort**, para depois manter só os N **utilizadores** necessários, algo que fica guardado numa **GList*** de nome **aux**.

Criamos uma **GHashTable*** que vai conter todos as tags. De seguida, percorremos a **GList*** **posts_perguntas** que só contém as N **perguntas** da lista **aux**.

Caso o **post_pergunta** esteja entre as datas passadas como argumentos, colocamos numa **GList*** denominada **tags** as tags desse mesmo **post_pergunta**. Vamos então percorrer esta lista e procurar as tags desta lista na **GHashTable*** **todas_tags**.

Nesse instante, caso a tag não exista na lista é criada uma nova instância da estrutura **Tag_Unique** e são passados o **tag_name** e a **ocorrência**, que será igual a 1. Caso já exista, as **ocorrências** são incrementas em 1 valor.

De seguida, é ordenada a lista com as N **tags** de acordo com o número de ocorrências.

Para finalizar usamos a função **g_hash_table_lookup** para procurar o **id_tag** relacionado com o **tag_name** no ficheiro Tags.xml.

6.14 Clean

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

Função que liberta a memória da **TCD_community**. Por consequência, realiza o **free** da estrutura **utilizador**, utilizando a função da *glib*, **g_hash_table_foreach**. De seguida, libertamos a própria **GHashTable*** através da função **g_hash_table_destroy**. O mesmo sucede-se com a estrutura **posts** e **tag**. A estrutura **date_posts**, sendo uma **GList*** tem uma forma diferente de libertar a memória. Para tal ser efetuado vamos a todos os parâmetros da mesma e usamos a função **g_list_remove** para libertar a memória de todos os elementos

constituintes da lista em questão. De seguida, realizamos o `g_list_free` de forma a libertar a própria estrutura.

7 Conclusão

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

Tendo em conta os objetivos definidos, o grupo acha que os resultados principais foram obtidos. Além de aprender a trabalhar com ficheiros do tipo **XML**, aprendemos a trabalhar com a biblioteca **GLib**. Aliamos também uma evolução da nossa aprendizagem no tema do **encapsulamento**, da **modularidade** e da **abstração de dados**.

Outros objetivos alcançados são a resolução total das queries e a eficiência que se faz notar nestas mesmas, assim como a libertação total de memória alocada pelo programa.

De uma forma natural, o grupo sentiu dificuldades durante o projeto, sendo uma, por exemplo, a necessidade de alterar a base total do trabalho devido a termos inicialmente considerado a ordenação dos ficheiros, quando só nos faltava resolver a query 11.

8 Bibliografia

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

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