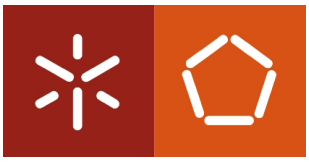
RELATÓRIO DO PROJETO EM C

Laboratórios de Informática III

Grupo 24:

Henrique Pereira (a80261)
Pedro Moreira (a82364)
Pedro Ferreira (a81135)

Universidade do Minho
Mestrado Integrado em Engenharia Informática



2018

Conteúdo

Listings

1 Introdução

No âmbito da unidade curricular Laboratórios de Informática III, do 2º ano do Mestrado Integrado em Engenharia Informática, foi-nos proposta a realização de um projeto que consiste no desenvolvimento de um sistema capaz de processar ficheiros XML, armazenando as várias informações utilizadas pelo Stack Overflow (uma das comunidades de perguntas e respostas mais utilizadas atualmente por programadores em todo o mundo). Além disso, após o tratamento da informação, o sistema teria de ser capaz de responder eficientemente a um conjunto de interrogações (explicitado na secção **??**). Esta aplicação teria que ser obrigatoriamente desenvolvida em *C*.

Ora, perante este enunciado, decidimos utilizar, na estruturação do nosso sistema, as definições da biblioteca GLib (versão 2.56.1), mais propriamente as GHashTables, as GTrees e os GArrays (e respetivas funções). As estruturas por nós definidas serão apresentadas e justificadas na secção **??**.

O grupo decidiu organizar o trabalho, separando o código em diferentes ficheiros, isto é, um ficheiro para cada estrutura por nós definida e utilizada (*structPost.c*,*structTag.c*,*structTCD.c* e *structUser.c*) e respetivas funções para interagir com as mesmas, um para o *parsing* dos ficheiros XML (*load.c*), um para cada interrogação (*query_*.c*), um para as funções auxiliares (*my_funcs.c*), e outro para a *main.c*. As funções de *init* e *clean* estão definidas no ficheiro *structTCD.c*.

Um ponto fulcral deste trabalho é conciliar a eficiência com o encapsulamento, sem comprometer nenhum destes.

2 Estrutura

Para a realização do projeto, tivemos de definir várias estruturas de dados em *C*, de maneira a respeitarmos o enunciado, no que toca aos tipos concretos e abstratos de dados. Foi-nos dada a seguinte definição abstrata de uma comunidade do Stack Overflow:

Listing 1: Definição da TAD_community

```
typedef struct TCD_community * TAD_community;
```

Assim sendo, tivemos como desafio criar a nossa própria definição concreta da comunidade. Após ponderarmos em grupo, tendo em conta as interrogações que nos apresentaram, a eficiência e o encapsulamento, decidimos organizar a nossa *TCD_community* da seguinte maneira (utilizando as definições da biblioteca GLib):

- Uma tabela de Hash para os Utilizadores
- Uma árvore binária balanceada para os Posts, ordenada por ordem cronológica inversa
- Uma tabela de Hash para acesso direto à data dos Posts pelo seu ID
- Uma tabela de Hash para as Tags

Listing 2: Definição da TCD_community

```
struct TCD_community{
        GHashTable* user;
        GTree* post;
        GHashTable* postAux;
        GHashTable* tags;
};
```

A nossa decisão de utilizar as tabelas de Hash vai de encontro ao propósito de aceder diretamente à informação por nós pretendida, no caso de um utilizador, a partir de um ID, no caso de uma Tag, a partir do seu nome. Porém, para os Posts, decidimos utilizar uma estrutura diferente, neste caso uma árvore binária balanceada, pois necessitávamos de uma ordem (neste caso cronológica) de recolha da informação dos Posts da estrutura que faz uma associação do tipo chave-valor, sendo a ordem dos elementos estabelecida pela comparação das chaves. No entanto, sendo os Posts uma unidade central no enunciado proposto, o grupo deparou-se com a necessidade de obter informação relativa a um certo Post, dado o seu *ID*, da forma mais eficiente possível. Claramente, realizar uma travessia (*in-order*) na árvore até encontrar o desejado não seria uma boa solução. De forma a otimizar o pretendido, foi adicionada à estrutura *TCD_community* uma tabela de Hash

cuja chave corresponde ao ID de um Post e o valor armazenado ao *timestamp* do mesmo, considerado até ao milissegundo. Deste modo, por comparação de chaves da árvore seria possível efetuar uma procura binária. Contudo verificou-se a exitência de diferentes posts cujos *timestamps* eram iguais, e a estrutura usada para os armazenar não contempla chaves repetidas. Assim, foi criada a estrutura *postKey*, também esta tornada opaca e devidamente encapsulada, apenas contendo a data e o ID de um Post, que passou a ser usada como chave de inserção na árvore. Desta forma garantiu-se não só a ordenação pretendida como a unicidade de cada uma das chaves. No caso de dois Posts terem o mesmo *timestamp*, considerou-se como sendo maior o Post com menor ID. Para efeitos de procura de um Post, foi criada a função *getPost*, cuja execução se divide nas duas tarefas descritas anteriormente: dado o ID de um Post, criar uma variável do tipo *postKey*, com o ID e o *timestamp* presente na tabela de Hash *postAux* e, de seguida, proceder a uma procura binária na árvore dos Posts.

Por sua vez, definimos também uma estrutura para os utilizadores, para os Posts e para as Tags, de forma a permitir que a informação útil contida em cada *row* dos ficheiros XML fosse guardada em variáveis representativas das entidade aqui enumeradas e posteriormente inseridas nas tabelas/árvore acima descritas. As estruturas descritas no *Listing* **??** foram tornadas opacas, sendo apenas possível a sua criação, alteração ou consulta da informação por estas contidas, através de funções definidas no ficheiro que contém a declaração da estrutura e cuja assinatura tenha sido exportada no respetivo *header file*. Deste modo, não só garantimos um acesso controlado aos campos internos de cada estrutura, como dotamos o projeto de um maior nível de abstração e modularidade.

Listing 3: Definição de estruturas internas

```
struct user{
        long id; // ID do utilizador
        char* display_name; // username
        int n_posts; // numero total de posts
        int reputacao; // reputacao do utilizador
        char* short_bio; // descricao do utilizador
        GArray* userPosts; // array dinamico com os posts do
            utilizador
};

struct post{
        long id; // ID do post
        char* titulo; // titulo do post
        long owner_id; // ID do criador do post
        int owner_rep; // reputacao do criador do post
        int type_id; // tipo do post
        long parent_id; // ID do "pai" do post (se este for
            resposta)
        char* data; // data do post
```

```
        GArray* tags; // IDs das tags usadas no post
        int score; // score do post
        int n_comments; // numero de comentarios do post
        int n_respostas; // numero de respostas do post (se
            este for pergunta)
};

struct tag{
        char* name; // nome da tag
        long id; // ID da tag
};
```

3 Interrogações e abordagem

Nesta secção, iremos explicitar a nossa abordagem às interrogações descritas no enunciado, mostrando algumas partes do código que achamos importante incluir de forma a facilitar a justificação da abordagem referida. Como o encapsulamento e a modularidade foram algo que sempre quisemos preservar ao longo do desenvolvimento do projeto, separamos a resolução de cada *query* por diferentes ficheiros, de modo a que cada *query_ *.c* contenha uma função auxiliar à resolução da devida interrogação. Tendo em mente que as implementações propostas incidem sobre os dados presentes na *TCD_ community*, a resolução das *queries* consiste na invocação feita no ficheiro que contém a definição da estrutura, da respetiva função auxiliar. Esta úlima recebe como argumentos partes da estrutura *TCD_ community* de que necessita, assim como os argumentos passados à função principal.

3.1 Info From Post

A primeira interrogação tinha como objetivo definir uma função que retornasse o título e o nome do autor dado um ID de um Post. Se o Post for uma resposta, a função deverá retornar informações (título e utilizador) da pergunta correspondente.

Listing 4: Query 1 - assinatura da função auxiliar

```
STR_pair info_from_post_aux(GTree* com_post, GHashTable*
    com_postAux, GHashTable* com_user, long id);
```

Ora, dada a estrutura escolhida pelo grupo, apenas foi necessário procurar o Post com o dado ID e, no caso desse Post ser uma resposta, procurar pela respetiva pergunta, em ambos os casos, através da função *getPost*. De seguida, foi necessário procurar a informação relativa ao criador do Post, através da função *getUser* e colocar no *STR_ pair* a devolver o título do Post e o nome do autor, como pretendido.

3.2 Top Most Active

Na segunda interrogação, era-nos pedida uma função que criasse uma lista com os N utilizadores mais ativos, isto é, com o maior número de Posts. Essa lista seria, portanto, uma *LONG_ list*.

Listing 5: Query 2 - assinatura da função auxiliar

```
LONG_list top_most_active_aux(GHashTable* com_user, int N);
```

Para tal, consideramos todos os utilizadores sobre a forma de um array dinâmico, obtido através da função *usersHashToGArray*, ao qual é aplicado a função *g_ array_ sort*, passando-lhe como argumento uma função que permite

comparar o número de Posts efetuados por dois Users. A parte final da resolução desta *query* corresponde a inserir na *LONG_list* a devolver os IDs dos Users presentes nas N posições iniciais do array dinâmico.

3.3 Total Posts

Nesta terceira *query*, a função que desenvolvemos tinha como objetivo obter o número total de posts (identificando perguntas e respostas separadamente) efetuados num dado período de tempo.

Para tal, bastou-nos percorrer a árvore dos Posts e comparar as datas de cada post com o intervalo de tempo dado, incrementando o número de perguntas ou de respostas, caso este estivesse dentro do intervalo. A função disponibilizada pela *GLib* para percorrer árvores binário do tipo *GTree* permite aplicar uma função passada como parâmetro a cada par chave-valor encontrado. O tipo de retorno desta função é um *gboolean*, sendo que, quando é devolvido *TRUE*, a travessia é interrompida. Esta caraterística foi também determinante na adoção da estrutura escolhida para armazenar os Posts, uma vez que em perguntas cuja contabilização de um dado Post na resposta dependa da sua ocorrência ter sido dentro ou fora de um dado intervalo de tempo, podemos minimizar o número de iterações desnecessárias sobre a árvore. Isto é, aquando da travessia da árvore, a partir do primeiro Post cuja data seja anterior ao limite inferior do intervalo do tempo, todos os outros Posts que seriam visitados posteriormente não fariam, certamente, parte da resposta final. Logo não precisam de ser visitados.

Listing 6: Query 3 - mecanismo de diminuição de iterações desnecessárias

```
int dateCheck = comparaDatas(date_begin, date_end, post_date);

(...)

if(dateCheck == -1) return TRUE;
```

Listing 7: Implementação da função comparaDatas

```
int comparaDatas (char* begin, char* end, char* post_date){

        if(strcmp(post_date,end)>0)
                return 1; // data do Post ainda nao entrou no
                    intervalo de tempo
        if(strcmp(begin,post_date)>0)
                return -1; // data do Post fora do limite
                    inferior(Return True na travessia)
        else
                return 0; // data do Post esta dentro do
                    intervalo de tempo
}
```

Este mecanismo fui útil, não só na resolução da *Query 3*, mas também, nas *queries* 4, 6, 7, e 10.

3.4 Questions with Tag

A quarta interrogação pedia que criássemos uma lista com os IDs das perguntas que, num dado intervalo de tempo, contivessem uma certa *tag*, sendo que essa lista teria de ser ordenada em cronologia inversa.

Nós abordamos esta *query* criando um array dinâmico, no qual colocámos os IDs das perguntas que, ao percorrer a árvore, verificamos estar dentro do intervalo de tempo dado e que continha a determinada *tag*. Para ser possível fazer tal verificação foi necessário obter o ID representativo da *tag* passada como argumento, conseguido através da execução das funções *getTag*(procura, pelo nome de uma *tag*, na tabela de Hash onde se encontra armazenada informação relativa às *tags*) e *getTagID*.

Após isso, bastou apenas passar a informação presente no array (já ordenado por ordem cronológica inversa) para uma *LONG_list*:

Listing 8: Query 4 - array dinâmico para LONG_list

```
for(int i=0; i<size; i++){
                long id = g_array_index(questionsID,long,i); //
                    questionsID: array dinamico
                set_list(result, i, id); //result: LONG_list a
                    devolver
}
```

3.5 Get User Info

Para esta quinta *query*, como a nossa estrutura de utilizadores continha ambas as informações necessárias, foi suficiente ordenar em cronologia inversa as publicações do utilizador em causa e converter os dados necessários para a estrutura *USER*, como pedia a interrogação. Para obter os Posts realizados pelo utilizador é usada a função *getClonedUserPosts*, que faz uma shallow-copy do *GArray* que contém os posts do utilizador. Desta maneira, podemos reordenar por data o array dinâmico sem que essa alteração tenha repercussões na informação presente na *TCD_community*.

Listing 9: Query 5 - preservação do encapsulamento

```
(...)
User user = getUser(com_user, id); // id - passado como
    parametro a query

        if(user!=NULL){
                GArray* userPosts = getClonedUserPosts(user);
```

```
                if(userPosts){
                        g_array_sort(userPosts,(GCompareFunc)
                            sortByDate);
(...)
```

3.6 Most Voted Answers

A sexta interrogação tinha como objetivo dar os IDs das N respostas com mais votos, em ordem decrescente do número de votos, num dado intervalo de tempo. Para tal, o grupo decidiu percorrer a árvore, inserindo num array dinâmico todos as Respostas que estivessem enquadradas na cronologia dada, ordenando posteriormente o array pelo *Score* de cada Post lá inserido, sendo que este é a diferença entre os *UpVotes* e os *DownVotes*.

Listing 10: Query 6 - função de ordenação

```
int sortByScore(Post *a, Post *b){
        int score_a = getPostScore(*a);
        int score_b = getPostScore(*b);
        return score_b - score_a;
}
```

3.7 Most Answered Questions

A sétima interrogação é muito idêntica à sexta *query*. Pedia-nos que criássemos uma lista ordenada com as perguntas mais respondidas, num dado intervalo de tempo.

Assim sendo, a nossa abordagem foi também muito semelhante, modificando apenas a inserção de Respostas no array dinâmico (passamos a inserir as Perguntas) e a ordenação deste pelo número de respostas, em vez do seu *Score*.

3.8 Contains Word

O objetivo da oitava *query* era, dada uma palavra, devolver uma lista com os IDs das N perguntas cujos títulos a contenham, ordenados por cronologia inversa.

Para resolver esta interrogação, tivemos que percorrer a árvore dos Posts, inserindo num array dinâmico os IDs das perguntas cujos títulos contenham a palavra passada como argumento. Para verificarmos tal situação, utilizamos a função *strstr*.

Listing 11: Query 8 - comparação do título e da palavra dada como argumento

```
if(strstr(titulo,word)!=NULL ){ // word: palavra dada como
    argumento
```

```
        g_array_append_val(postArray,post); // funcao que
            coloca o post no final do array dinamico
}
```

3.9 Both Participated

Nesta nona interrogação, tivemos o "desafio"de, dados os IDs de dois utilizadores, devolver as últimas N perguntas (cronologia inversa) em que participaram dois utilizadores específicos. Estes podem interagir de três diferentes formas: um deles faz a pergunta e o outro a resposta (e vice-versa) e ambos responderem a uma mesma pergunta.

A abordagem que considerámos mais eficiente foi a seguinte:

1. Carregar as informações dos dois utilizadores dados como argumento, em particular o *GArray* contendo os Posts realizados por cada um dos utilizadores;
2. Identificar o array com maior tamanho e inserir os dados nele contidos num tabela de Hash, de forma a posteriormente ser possível iterar sobre o outro array, já ordenado, por cronologia inversa, e apenas contendo perguntas, com o objetivo de verificar existência de cada elemento do segundo array na tabela de Hash.
3. Selecionar as N primeiras perguntas do novo array com as perguntas em comum.

Listing 12: Query 9 - identificação dos Posts comuns aos dois utilizadores

```
GHashTable* toSearch = gArrayToHash(toConvert);

GArray* result_aux = g_array_new(FALSE, FALSE, sizeof(long));

for (int i=0; i < toTraverse->len; i++) {
        post = g_array_index(posts1, Post, i);
        id = getPostID(post);

        int* check = g_hash_table_lookup(toSearch, (gpointer)id
            );

        if(check){
                g_array_append_val(result_aux,id);
                g_hash_table_remove(toSearch, (gpointer)id);
        }
}
```

3.10 Better Answer

A décima interrogação pedia que, dada uma determinada pergunta, obtivéssemos a melhor resposta, através da seguinte função de média ponderada:

```
(score*0.45) + (reputation*0.25) + (votes*0.2) + (comments*0.1)
```

Ora, como a diferença entre os *UpVotes* e os *DownVotes* é igual ao Score, decidimos não utilizar o ficheiro Votes.xml e utilizar o Score presente no ficheiro dos Posts. Os restantes parâmetros encontram-se guardados na estrutura por nós definida após ser executado o *load*.

Para conseguirmos responder ao que nos foi pedido, tivemos de verificar se o ID do Post dado correspondia a uma pergunta e, se assim fosse, iterar a árvore dos Posts para verificar quais são respostas para essa pergunta e selecionar, então, a resposta com a maior pontuação, calculada pela média acima referida.

3.11 Most Used Best Rep

Nesta décima primeira (e última) *query*, era pretendido que, dado um intervalo arbitrário de tempo, fosse possível obter os IDs das N *tags* mais utilizadas pelos N utilizadores com melhor reputação.

Para chegar a tal resultado, foi preciso ordenar os utilizadores por ordem decrescente de reputação e, posteriormente, percorrer os N utilizadores com melhor reputação. Para cada um deles, fomos ao seu array de Posts e adicionamos as tag usadas em cada um dos posts a uma tabela de Hash, cuja chave é o ID da *tag* e o valor corresponde ao número de ocorrências dessa *tag*. Após este processo, passamos a informação contida na tabela de Hash para um array dinâmico. Por último, foi apenas necessário ordenar esse vetor pelo número de ocorrências e retirar daí os IDs das N *tags* mais usadas para uma *LONG_ list*.

Listing 13: Query 11 - registo da ocorrência de uma *tag* na tabela de Hash

```
(...)
for(int i = 0; i<postTags->len; i++){
                long next_tag = g_array_index(postTags, long, i
                    );

                int* num_ocor = g_hash_table_lookup(tagsId , (
                    gpointer)next_tag);

                if(!num_ocor) {
                        int* ocor = malloc(sizeof(int));
                        *ocor = 1;

                        g_hash_table_insert(tagsId, (gpointer)
                            next_tag, ocor);
                }
                else (*num_ocor)++;
        }
```

4 Conclusão

Este projeto foi extremamente enriquecedor para o grupo, uma vez que nos permitiu trabalhar com um tipo diferente de dados, o XML, e nos ensinou a trabalhar com ele. Tivemos que nos esforçar bastante para conseguirmos atingir os objetivos estabelecidos, tendo sempre em atenção o encapsulamento do código e a sua eficiência. O trabalho permitiu-nos também aprender a desenvolver as estruturas de que precisamos, consoante as dificuldades, exigências e necessidades que nos foram sendo impostas com o desenrolar do projeto.

Os resultados que obtivemos, ao correr o programa, foram iguais aos dos testes fornecidos pelos docentes, pelo que podemos concluir que a nossa implementação está correta. Além disso, achamos que cumprimos o objetivo de criar um programa eficiente, visto que os tempos que obtivemos para cada interrogação, utilizando os *dumps* do Ubuntu e os mesmos parâmetros utilizados pelos professores nos testes, foram os seguintes (aproximadamente):

Tabela 1: Tempos de Execução - input nº 1

Interrogação:	Tempo (em segundos):
1	0.000024
2	0.143056
3	0.047433
4	0.161335
5	0.005250
6	0.039786
7	0.062277
8	0.110665
9	0.000371
10	0.129559
11	0.159771

Tabela 2: Tempos de Execução - input nº 2

Interrogação:	Tempo (em segundos):
1	0.000035
2	0.141657
3	0.133918
4	0.136191
5	0.006277
6	0.096734
7	0.126107
8	0.117705
9	0.000102
10	0.135200
11	0.162032

A nossa função *clean* também está a funcionar como o esperado, como se pode comprovar pelo relatório do *valgrind*:

```
==1969== HEAP SUMMARY:
==1969==     in use at exit: 92,784 bytes in 37 blocks
==1969==   total heap usage: 19,704,863 allocs, 19,704,826 frees,
1,352,717,691 bytes allocated
==1969==
==1969== LEAK SUMMARY:
==1969==    definitely lost: 0 bytes in 0 blocks
==1969==    indirectly lost: 0 bytes in 0 blocks
==1969==      possibly lost: 0 bytes in 0 blocks
==1969==    still reachable: 92,784 bytes in 37 blocks
==1969==         suppressed: 0 bytes in 0 blocks
==1969== Reachable blocks (those to which a pointer was found) are not shown.
==1969== To see them, rerun with: --leak-check=full --show-leak-kinds=all
==1969==
==1969== For counts of detected and suppressed errors, rerun with: -v
==1969== ERROR SUMMARY: 0 errors from 0 contexts (suppressed: 0 from 0)
```

Em suma, tendo em conta a eficiência e os resultados que obtivemos, podemos concluir que conseguimos atingir as metas propostas com este projeto, dado que os tempos de execução, tanto do *load* como das interrogações, foram bastante positivos. Com os *getters* e *setters* por nós definidos para as nossas estruturas, conseguimos sempre, com a modulação e separação dos ficheiros que fizemos, respeitar o encapsulamento da informação.