Relatório do Trabalho Prático de LI3

Grupo 42

João Queirós A82422

José Costa A82136

Luís Alves A80165

Miguel Carvalho A81909

5 de Maio de 2018

Conteúdo

Capítulo 1

Introdução

1.1 Contextualização

Este projeto foi desenvolvido no âmbito da Unidade Curricular de *Laboratórios de Informática III*, no qual analisaremos e processaremos informação do ***Stack Overflow***, que nos permitirá criar funções...

1.2 Objetivos

Ao longo do semestre, a realização deste projeto levará a um reforço de competências ganhas em unidades curriculares de semestres anteriores, como *Programação Imperativa*, *Arquitetura de Computadores* e *Algoritmos e Complexidade*, assim como a sedimentação dos conhecimentos adquiridos em *Programação Orientada aos Objetos*, aplicando-os na construção de um trabalho concreto, aumentando as nossas capacidades de comunicação e trabalho em grupo. Através da utilização de um sistema de controlo de versões, ***GitHub***, fomos deste modo incentivados a criar boas práticas de trabalho.

Capítulo 2

Abordagem Inicial

2.1 Estruturas de Dados

Inicialmente, a preocupação do grupo centrou-se na implementação de estruturas para alocar os dados de uma forma eficiente, de modo a permitir um rentável processamento dos mesmos.

A nossa primeira abordagem consistiu em implementar uma *AVL Tree*, por ser uma estrutura ordenada e eficiente em termos assintóticos para o tratamento dos dados. No entanto, devido a uma melhor API, assim como um tempo de execução inferior dadas as circunstâncias, decidimos alterar a estrutura para uma *Sequence*. Esta contém os Posts ordenados pela sua data (por ordem crecente) de modo a permitir uma procura eficiente.

Implementamos também uma *Hash Table*, deste modo é possível utilizar a Hash para uma procura rápida de um post específico por ID, e a Sequence para aplicar uma ou mais funções a um bloco de Posts.

Por último, criamos ainda uma *Heap* que serve para guardar os Users, sendo que estes são guardados em duas Heaps ordenadas de forma diferente, uma delas ordenada pela reputação, e a outra pelo número de posts (ambas ordenadas por ordem decrescente).

2.2 Parsing

O nosso *parsing* consistiu em colocar os Posts numa Hash Table, assim como numa *Sequence*.As Tags são também colocadas numa Hash Table, bem como os Users, estes posteriormente colocados em duas heaps, uma delas ordenada pela reputação, e a outra pelo número de posts (ambas ordenadas por ordem decrescente).

Capítulo 3

Dados

3.1 Abstração

Após ser feito o *parsing*, era necessário criar vários tipos abstratos de dados que contivessem a informação recebida dos ficheiros.

Assim sendo, recorremos ao ***GLib***, onde encontramos implementações gerais de *structs*, que ao longo do projeto concretizamos para satisfazerem a resolução dos problemas propostos.

Além deste suporte, também na implementação do código se encontram funções que se generalizam para várias estruturas, nomeadamente as que têm como parâmetro um apontador, ou seja, existe a possibilidade da função invocar qualquer estrutura implementada no projeto.

3.2 Modularização

Para aplicar modularização funcional ao projeto desenvolvido foi decidido em conformidade criar um ficheiro para cada estrutura de dados (isto é, foi implementada uma *Hash Table* no ficheiro `hash.c`, uma *Heap* no ficheiro `heap.c`, etc). Posteriormente, foi ainda feito um *union* no ficheiro `post.c`, que separa para o ficheiro `question.c` os *posts* que são perguntas e para `answer.c` os que são respostas.

Todos estes ficheiros são incluídos no header de `interface.c`, ficheiro esse onde se encontram implementadas todas as *queries* propostas e, logicamente, usando todas as estruturas que foram importadas.

Capítulo 4

Queries

4.1 Query 1

São invocadas as funções `tad_getPostsByHash` e `tad_getUsersByHash` para colocar, respetivamente, posts e utilizadores numa *Hash Table*, sendo em seguida verificada a existência do post.

Caso se trate de uma pergunta, encontra-se o ID do criador da mesma (`post_getCreatorID`), que é posteriormente pesquisado na *Hash Table* de utilizadores. Por fim, é criado o par de *strings* que devolverá o título e nome de utilizador do autor do *post*.

A estratégia caso o post se trate de uma resposta é obter a pergunta onde está inserida a mesma (através de `post_getParent`). Daí em diante, o processo é análogo ao feito com as perguntas.

4.2 Query 2

Os utlizadores são guardados numa *Heap* ordenada pelo número de *posts* feitos. É-lhe feito um *clone* que aplica uma cópia de bloco de memória (`memcpy`) para se ajustar ao tamanho dos utilizadores.

Para correr o ciclo que passa pelos N utilizadores mais ativos é calculada a variável `maxSize` que, claro está, será o tamanho da lista de utilizadores devolvida. Lista essa que é gerada retirando ordenadamente da *Heap* cada utilizador, encontrando o seu ID e inserindo-o por fim na lista referida. Após todas as iterações é libertado o espaço ocupado pela *Heap* através da função `heap_free`.

4.3 Query 3

Começamos por criar um *Long Pair* $(0, 0)$ para contar o número de perguntas e respostas e criamos a sequência com todos os posts. A seguir, chamamos a função `seq_inBetweenDates`, que vai aplicar uma função a todos os posts entre as datas recebidas. Essa função vai verificar se os posts são perguntas ou respostas, e incrementa o contador em conformidade.

4.4 Query 4

-M Os posts são guardados numa *Hash Table*. Utilizamos a função `seq_inBetweenDates` que vai aplicar uma função a todos os posts da Hash entre as duas datas. A função vai inserir os posts dinamicamente numa lista caso estes sejam perguntas e contenham a tag fornecida. No final de todos os posts serem inseridos, o espaço não ocupado é libertado e a lista é invertida.

4.5 Query 5

Tal como na **Query 1**, é guardada a informação dos utilizadores numa *Hash Table*, que é usada para obter a informação do utilizador com o ID invocado como argumento.

Testado o caso em que o utilizador não existe, através das funções `long_user_get10LastPosts` e `long_user_getBio` (ambas recebem um utilizador) é obtido o perfil e os ID's dos seus dez últimos posts, do mais recente para o mais antigo.

É, então, criado um utilizador com as informações mencionadas (através da função `create_user`) que é devolvido como resultado da Query.

4.6 Query 6

Chama a função `seq_inBetweenDates` que vai aplicar uma função a todos os posts entre as datas e se fôr uma resposta coloca o seu ID, se possível, ordenadamente numa lista de tamanho fixo N (ordenadas decrescentemente pelo número de votos).

4.7 Query 7

-M Chama a função `seq_inBetweenDates` que vai aplicar uma função a todos os posts entre as datas e se fôr uma pergunta coloca o seu ID, se possível, ordenadamente numa lista de tamanho fixo N (ordenadas decrescentemente pelo número de respostas).

4.8 Query 8

Para esta **Query** foi implementada uma função auxiliar que verifica se uma determinada palavra está presente no título de um post.

A estratégia usada para resolver a questão foi colocar os dados num *Array Pointer* de tamanho 2, guardando apontadores de uma *Sequence* no primeiro e da palavra a ser testada. Em seguida, são percorridos os posts numa *Hash table* para colocar ordenadamente na *Sequence* os que contêm a palavra procurada.

É, então, inicializado um *Array* que, através da função `seq_getNFirstPost`, passa a ter o mesmo conteúdo da *Sequence* descrita anteriormente. Por fim, é feito um ciclo para criar a lista a ser devolvida (a partir do *Array*) e liberta-se o espaço das estruturas geradas.

4.9 Query 9

As funções auxiliares desta **Query** (`both_participated_addPostsToHash` e `both_participated_addCommonToSeq`) são usadas para adicionar ID's de *Posts* a uma *Hash table* ou a uma *Sequence*, enquanto que a função `both_participated_cleanMemory` liberta o espaço de todas as estruturas usadas na função principal.

Na referida função principal são encontrados os utilizadores segundo os ID's que são passados como parâmetros da mesma. Testadas as suas existências, é colocada numa *Hash table* os *Posts* do primeiro utilizador e em duas *Sequence* os N primeiros ID's de posts dos dois utilizadores. Com a primeira função auxiliar, coloca-se numa outra *Hash table* os ID's de posts do Utilizador 1. Logo após invoca-se a função `both_participated_addCommonToSeq` descrita acima, que encontra o ID de um *Post* e acrescenta-o à *Sequence* se o mesmo ainda não se encontrar nela e estiver na *Hash table* do Utilizador 1.

Por fim, é utilizado o mesmo processo final da **Query 8** que gera a lista a partir da *Sequence* que contém os ID's de todos os *Posts* onde ambos os utilizadores participaram.

4.10 Query 10

É encontrado o *Post* procurando numa *Hash table* o ID da pergunta. São então percorridas todas as respostas à pergunta e, para cada uma delas, é calculado o seu *score*. Através de um algoritmo de cálculo de elemento máximo, é colocado na variável a ser devolvida pela função o ID da resposta.

4.11 Query 11

Retiramos da heap dos users os N melhores qualificados e colocamo-los numa lista. Chamamos a função `seq_inBetweenDates` que aplica uma função a cada post entre as datas recebidas. A função verifica se o post pertence a um dos N utilizadores e caso pertença aplica uma outra função ás tags que a associa ao seu ID e insere-o numa Hash que, a cada ID(key) associa um contador, e caso o ID já exista, incrementa o contador da respetiva tag. Finalmente, percorremos a hash e inserimo-los por ordem decrescente de aparências na lista a retornar.

Capítulo 5

Estratégias de otimização

Não foi utilizada nenhuma API para melhorar e influenciar os tempos de execução das *Queries* (como por exemplo o ***OpenMP***.

Desde cedo que o grupo se focou em criar estruturas de dados com complexidade linear ou logarítmica(*Hash tables* e *AVL Trees*) de modo a manter sempre um bom desempenho na execução das funções criadas. Beneficiamos também das implementações já construídas fornecidas pelo ***GLib*** (como foi dito anteriormente) que contribuíram para a implementação de todo o código com um desempenho considerável. Deste modo, toda a otimização feita pelo grupo baseou-se em gerir o código da maneira mais limpa possível, como retirando qualquer tipo de *"ciclos encavalitados"* que geram complexidades polinomiais e evitando a cópia de conteúdo, ou seja, trabalhando sobretudo com *pointers*.