Universidade do Minho, Departamento de Informática
MIEI, 2020-2021

Laboratórios de Informática III

Relatório do Trabalho Prático de C

Autores

Alexandre Silva Martins A93315
Luís Francisco Ferreira Faria A93219
Tiago André Leça Carneiro A93207

Introdução

Para este projeto, foi pedido que desenvolvêssemos um programa capaz de gerir informação fornecida, através de ficheiros, sobre utilizadores, negócios e reviews.

Um dos aspectos mais complicados no desenvolvimento deste tipo de projetos é a escolha das estruturas de dados. Algumas estruturas favorecem a velocidade de carregamento dos dados, enquanto outras favorecem a leitura, em suma, todas têm desvantagens e vantagens. Dito isto, o nosso grupo preferiu priorizar a velocidade na execução das queries, organizando maior parte dos dados na leitura, tornando esta consideravelmente longa.

As estruturas de dados utilizadas foram: HashTables e Arrays Dinâmicos.

Módulos

- ## Users

As chaves principais deste módulo são as estruturas:

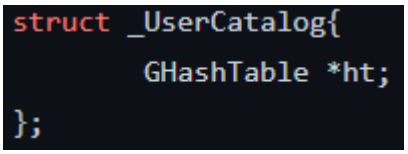
- **user** (Users.c), em que foram usados **arrays de chars** para armazenar o **nome** e os **ids**, tanto do utilizador, como dos seus **amigos** que foram armazenados recorrendo a **arrays dinâmicos**.

```
struct user {
    char *user_id;
    char *name;
    GPtrArray *friends;
};
```

- **compUser** (userCatalog.c), estrutura utilizada para complementar as informações presentes na estrutura **user,** adicionando uma bool **'international'** de modo a **facilitar a contagem** de utilizadores que fizeram reviews de negócios a nível internacional (**query 7**), e um **array dinâmico** que contém o **id das reviews** feitas por este utilizador, permitindo assim conectar o módulo das reviews com o dos users.

```
struct compUser{
        User u; //User
        bool international; //Visitou 1 ou mais estados
        GPtrArray *reviews;     //Array dinamico com review ids
};
```

- UserCatalog (userCatalog.c), esta estrutura consiste numa **HashTable** onde armazenamos **compUser**s, sendo a **chave o id** de cada utilizador. Escolhemos esta estrutura de dados porque nos **permite acessos constantes** às informações de um utilizador específico caso seja fornecido o id deste, como por exemplo na **query 4**. Pelo facto de só utilizarmos uma **HashTable**, **perdemos** um pouco de **eficiência** a nível de tempo na **query 7**, onde é necessário percorrer todos os utilizadores de modo a selecionar os que são **"internacionais"**.



A **API dos Users** possui apenas um construtor **‘str_to_user’** que **transforma uma string**, em que os componentes estão separados por ‘;’, **num user**. O facto de não poder utilizar outros delimitadores, acompanhado de não existirem setters, tornam a **API bastante limitadora**. A API possui gets de cada uma das componentes da estrutura **user**.

A **API do módulo userCatalog** consiste apenas no necessário para a execução das diversas queries como **“Init”** e **“Free”** do catálogo, e alguns gets e setters. A **falta** de um conjunto de **gets e setters** torna este módulo demasiado específico, tornando **difícil a sua reutilização**.

- **Negócios**

As estruturas usadas neste módulo foram:

- **business,** nesta estrutura, à semelhança da estrutura **user,** recorremos a arrays de chars para guardar as diversas informações, assim como um array dinâmico para guardar as categorias.

```
struct business {
    char *business_id;
    char *name;
    char *city;
    char *state;
    GPtrArray *categories;
};
```

- **compBusiness,** esta estrutura complementa **business,** acrescentado uma variável 'stars', onde é guardado o rating do negócio em termos de estrelas, assim como um array dinâmico onde são guardados os ids das reviews feitas a este negócio, conectando assim estes dois módulos.

```
struct compBusiness{
        Business business;  //Business
        float stars;            //Estrelas deste business
        GPtrArray *reviews; //Array dinamico com as reviews
        //GArray tem a componente 'len' que serve de review_count
};
```

- **BusinessCatalog,** este catálogo é constituído por 4 estruturas de dados:

1) **Array de 26 Arrays Dinâmicos de Business**, um array dinâmico por cada letra do alfabeto, esta estrutura foi utilizada com o intuito de um acesso mais rápido a todos os nomes começados por uma dada letra, como pedido na query 2;

2) **HashTable de CompBusiness** cuja key é o id do negócio, escolhida por permitir acessos em tempo constante à informação de um negócio, como é pedido na query 3;

3) **HashTable de Arrays Dinâmicos de CompBusiness,** estas estruturas são usadas para de um certo modo filtrar os negócios, permitindo acessos mais rápidos a negócios que têm um determinado parâmetro, i.e., pertencem a uma certa cidade ou têm uma certa categoria

(queries 5, 6 e 8). Usamos duas estruturas de dados deste tipo, uma em que as keys são o nome da cidade em que se encontra o negócio, e outra em que as keys são as categorias.

A **API dos Business** segue o mesmo padrão da API dos **Users,** tendo apenas um construtor que tem como argumento uma string e um conjunto de getters para todos os dados.

A **API do businessCatalog,** apenas fornece os métodos necessários para a resolução das queries.

Nota: As estruturas de CompBusiness compartilham os apontadores destes, e a estrutura compBusiness contém o apontador para a estrutura business.

```
struct structBusinessCatalog{
        BusinessByLetter bl;
        GHashTable* ht_businesses;
    GHashTable* ht_bus_categories;
    GHashTable* ht_bus_cities;
};
```

- **Reviews**
  As estruturas escolhidas para armazenar a informação relevante às reviews foram as seguintes :

  -**reviews** (Reviews.c), aqui armazenamos as **informações simples** provenientes da **primeira leitura** feita a cada linha de um ficheiro que contenha reviews.

```
struct review {
    char *review_id;
    char *user_id;
    char *business_id;
    float stars;
    int useful;
    int funny;
    int cool;
    char *date;
    char *text;
};
```

  -**book** (reviewsCatalog.c), esta foi a estrutura de dados que trata de armazenar a informação necessária para ser possível fazer a **query 9**. A nossa primeira abordagem para esta query foi utilizar uma **hashtable** que utilizava a palavra como chave de acesso e que armazenava os apontadores para as reviews que continham esta mesma.

```
typedef struct book{
        GHashTable * ht;
}*Book;
```

  No entanto, **decidimos explorar outro método** para resolver esta query, já que observamos que uma hashtable iria armazenar mais memória do que aquela estritamente necessária.Escolhemos uma estrutura composta por um array de **26 GPtrArrays** :

```
struct book{
        GPtrArray *dict[26];
};
```

Cada um destes arrays dinâmicos continha a seguinte estrutura de dados :

```
struct words{
        char * word;
        unsigned char size;
        GPtrArray * reviews;
};
```

O propósito desta abordagem era criar uma estrutura de dados que **ocupasse menos espaço que a hashtable com um custo no tempo de procura e escrita**, os 26 arrays foram criados para dividir as palavras pela sua 1ª letra, e o parâmetro size para apenas comparar a string do parâmetro word com a que pretendemos procurar caso o tamanho de ambas fosse igual.

No entanto, esta abordagem demonstrou-se como sendo **demasiado ineficiente na escrita e procura** de palavras e, por isso, acabamos por escolher a primeira abordagem como a definitiva para resolver a query 9.

- **ReviewsCatalog** (reviewsCatalog.c), que contém **2 hashtables**, uma que está guardada na estrutura **Book**, utilizada para resolver a **query 9**, e uma outra hashtable, que utiliza como **chave o id** da review e onde são armazenadas as **reviews**

```
struct _ReviewCatalog {
        GHashTable *ht;
        Book bk;
};
```

A **API dos reviews** segue o **padrão** das API's de tanto os users como dos business.

Já a **API do reviewsCatalog** contêm a função **'word_in_hash'**, responsável pela **query 9**, tal como funções para manusear as reviews que a hashtable da estrutura **_ReviewsCatalog** contêm

- **SGR**

O módulo **sgr** é o **foco principal** do projeto, sendo aqui o local onde **reunimos** todas as funções necessárias para atender às diversas **queries**.

A estrutura de dados sgr junta **3 estruturas** já anteriormente referidas, **UserCatalog**, **BusinessCatalog** e **ReviewCatalog**. Devido ao design das respetivas **API's** de cada um destes módulos conseguimos concluir a realização das diversas funções deste módulo **sem qualquer problema.**

- **SGR :**

```
struct sgr{
    BusinessCatalog bc;
    UserCatalog uc;
        ReviewCatalog rc;
};
```

- ## Stats

De forma a ocupar o **mínimo de memória** possível, optamos por **não criar** um módulo stats, mas sim **tratar de toda a informação** que seria aqui processada nos **diversos catálogos** já anteriormente referidos.

- ## Table

O módulo **table** é o local onde a **informação** obtida para a resolução das queries é **organizada** na estrutura de dados **Table** e onde ocorre a **manipulação** da mesma.

A estrutura de dados **Table** é uma lista de pointers de uma lista de strings, a "table", uma lista de strings que armazena os headers, uma lista de inteiros, o tamanho da maior string associada a cada header, e um inteiro, o número de headers.

Escolhemos esta estrutura **Table** porque ocupa, relativamente, **pouca memória**, facilita a execução das funções no módulo e para cumprir o requerimento de ser **reutilizável** pedido pelos docentes.

- **Table :**

```
#define MAX_INFOTYPES 5

struct Table {
    char infoTypes[MAX_INFOTYPES][20];
    int size[MAX_INFOTYPES];
    int nr_colunas;
    GPtrArray *table;
};
```

A funções prinicipais da **API do módulo table** são:

construct_Table – função que constrói a Table com os argumentos que recebe;

print_table – função que imprime a Table no terminal num determinado formato;

Proj;

Filter;

fromCSV;

ToCSV;

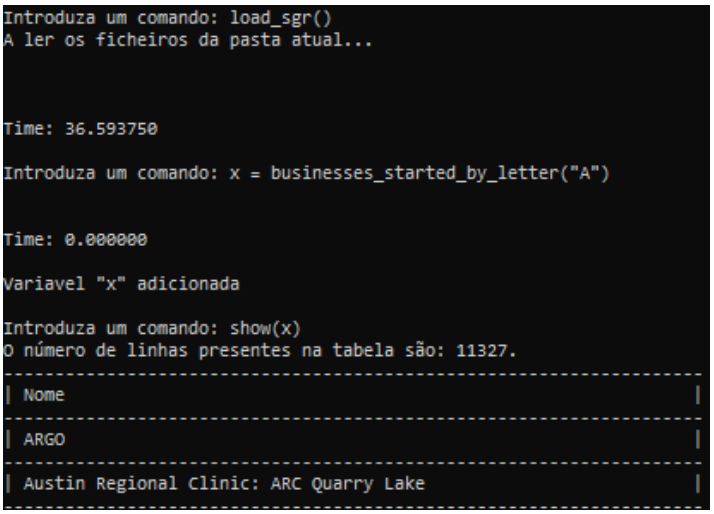
directAcess.

Para além destas funções também exitem funções auxiliares, getters, setters, clones, a **init_table** e a **free_table**.

Testes de Desempenho

Estes testes foram feitos recorrendo à biblioteca time.h, tendo isto em conta, o tempo medido corresponde ao tempo de uso de cpu.

- **Load SGR (Query 1) :** Usando os ficheiros fornecidos pelos professores, o users_full.csv, business_full.csv e reviews_1M.csv, verificamos que o tempo de carregamento foi de 36, 594 segundos, ocupando 7344,4 MBs de memória.
- **Businesses Started By Letter (Query 2) :** Este teste foi feito utilizando a letra “A”, ficando a tabela com 11327 nomes. Tempo de execução: Aparentemente 0.000000.



- **Business Info (Query 3):** Este teste foi feito utilizando o id “tCbdrRPZA0oiIYSmHG3J0w”. Tempo de execução: 0.000000
- **Businesses Reviewed (Query 4):** Este teste foi feito utilizando o id “1jXmzuIFKxTnEnR0pxO0Hg”, obtendo uma tabela com 26 negócios. Tempo de execução: 0.000000
- **Businesses With Stars And City (Query 5):** Este teste foi feito utilizando o número de estrelas 0 e a cidade Portland, obtendo uma tabela com 18022 negócios. Tempo da 1ª execução: 0.015625 ; Tempo da 2 ª execução: 0.000000. Isto acontece porque os dados não se encontram ordenados na primeira chamada da função para uma dada cidade.
- **Top Businesses By City (Query 6):** Este teste foi feito com o número 30, ficando com 8019 negócios. Tempo de execução: 0.031250
- **International Users (Query 7):** Número de users internacionais: 19159. Tempo de execução: 0.421875
- **Top Businesses With Category (Query 8):** Dando como argumentos o número 150000 e a categoria “Restaurants” obtemos um total de 50365 negócios nesta categoria. Tempo de execução: 0.046875
- **Reviews with words (Query 9) :** A palavra pesquisada foi “the”, obtendo como resultado 865819 reviews. Tempo de Execução: 0.312500

Conclusão

Na realização do projeto, priorizamos manter a modularidade e o encapsulamento dos módulos ao máximo e não haver memory leaks, de forma a usar relativamente pouca memória e executar tudo o mais eficiente possível, por isso, tudo foi feito internamente nos respetivos módulos, sacrificando um pouco do aspecto da reutilização do código.

Se pudéssemos fazer as API’s de forma diferente, faríamos de forma a que estas não fossem tão limitadoras, acrescentando os diversos construtores, setters e getters, permitindo assim que o código seja reutilizável.