Mestrado Integrado em Engenharia Informática

Laboratórios de Informática III

Trabalho C

Daniel Vieira A73974
Nadine Oliveira A75614
Duarte Freitas A63129

5 de Maio de 2018

Conteúdo

Capítulo 1

Introdução

Este trabalho está inserido na unidade curricular Laboratórios de Informática III, que tem como objetivo aplicar os diversos conhecimentos adquiridos até agora. Nesta primeira fase, o projeto foi implementado na linguagem C, foi devidamente particionado em modulos devidamente organizados, e foi sempre mantido o encapsulamento de dados.

Foi necessário desenvolver um sistema, capaz de processar vários ficheiros XML, que contêm várias informações presentes no Stack Overflow. Após estar efetuado o devido armazenamento da informação util retirada destes ficheiros, foi então iniciada o processo de respostas das interrogações propostas.

Capítulo 2

Estruturas

2.1 Users

2.1.1 Processamento da informação dos users

Após uma cuidosa análise do ficheiro xml fornecido relativo à informação dos *users*, foi efetuada uma filtragem e guardada na estrutura, User, apenas a informação necessária para responder às interrogações propostas. Para tal, os atributos selecionados foram: *Id, Reputation, DisplayName, AboutMe*. Para além disto, foi adicionado ainda, um campo extra à estrutura, *countPosts*, onde será guardado o respetivo número de posts do user em questão.

Para armazenar esta informação, foi criada uma estrutura *struct UserAVL*, em que cada nodo da árvore é um apontador para uma *struct User*. Foi escolhida uma avl ordenada pelo id do user, visto que, para as interrogações propostas este é o que oferece mais rapidez em termos de tempo de pesquisa (pior caso: log2 N, N=número de nós), melhorando assim a eficiencia das queries que a usam.

```
    typedef struct User{
    long idUser;
    long reputation;
    char* about;
    char* name;
    int countPosts;
} *User;

typedef struct UserAVL{
    User users;
    struct UserAVL* esq;
    struct UserAVL* dir;
    int altura;
} *UAVL;

typedef struct UserLL{
    User users;
    struct UserLL* prev;
```

```
    struct UserLL* next;
    int altura;
} *ULL;
```

2.2 Posts

2.2.1 Processamento da informação dos posts

Para os *posts*, foi necessário guardar uma quantidade superior de informação, tendo sido selecionados os seguintes atributos como relevantes: *Id, PostTypeId, CreationDate, OwnerUserId, Title, Tags, AnswerCount, CommentCount, ParentId, FavouriteCount, Score*. Para além desta informação recolhida diretamente do ficheiro xml, foi ainda adicionado uma informação extra considerada necessária, o número de votos que cada post possui.

Para armazenar a informação relativa aos posts, foi usada também uma avl ordenada por id de post.

```
    typedef struct Post{
    long idPost;
    int postType;
    Date creationDate;
    char* title;
    long parentId;
    char* tags;
    long score;
    long favCount;
    long owner;
    long nVotes;
    int comentCount;
    int answerCount;
} *Post;

typedef struct PostAVL{
    Post post;
    struct PostAVL* esq;
    struct PostAVL* dir;
    int altura;
} *AVLT;
```

2.3 Tags

2.3.1 Processamento de informação das tags

Para o ficheiro *tags*,e de acordo com o que é pretendido no enunciado, foi necessário guardar o id da tag, o nome, e foi ainda adiconado um *long* count, que inicialmente se encontrará como valor 0, porém será necessário posteriormente. Tudo isto estará guardado numa estrutura a que daremos o nome de *Tag*.

Para processar essa informação, foram criadas duas estruturas diferente. Uma lista ligada, *TagLL*, e uma avl *TagAVL*, em que cada nodo da lista tem um apontador para as *struct Tag* guardadas anteriormente.

```
typedef struct Tag{
long idTag;
    char* name;
    long count;
} *Tag;

typedef struct TagLL{
    Tag tag;
    struct TagLL* next;
} *TagLL;

typedef struct TagAVL{
    Tag tag;
    struct TagAVL* esq;
    struct TagAVL* dir;
    int altura;
} *TAVL;
```

2.4 Votes

Para o ficheiro *Votes.xml*, foi necessário guardar apenas o id do post a quem o vote se referia, e ainda o tipo do voto.

A informação recolhida, foi devidamente armazenada numa avl, ordenada por idPost. Porém foi adicionada uma informação considerada útil para o futuro, a classificação total do post, assim, durante a inserção na avl dos votos, sempre que era inserido um novo voto, era verificado se o id do post já se encontrava na avl, e se assim o fosse, a classificção total era incrementada ou decrementada, de acordo com o valor do *voteType* do voto a inserir.

```
typedef struct Vote{
    long postId;
    int voteType;
} *Vote;

typedef struct VoteAVL{
    long postId;
    int classPost;
    struct VoteAVL* esq;
    struct VoteAVL* dir;
    int altura;
} *VAVL;
```

2.5 TCD_Community

Depois de muita análise e pensando sempre em obter a melhor eficiencia possível, foi obtida esta estrutura que permite obter resposta a todas as interrogações apresentadas, bem como obter uma boa eficiencia para grande parte delas.

2.5.1 struct UserAVL - users

Foi criada uma *struct UserAVL* onde está armazenada toda a informação dos users, ordenada por id de user. Para além disso, foi introduzida uma variável *usersNumber*, que possui o número de users introduzidos na avl.

2.5.2 struct PostAVL - respostasByParent

Esta estrutura foi adicionada como suporte para dar resposta à interrogação número 10. Nesta estrutura são armazenados todas as respostas, ordenadas por *parentId*, porém caso haja mais que uma resposta para a mesma pergunta (*parentId* = id da pergunta), é guardada a resposta com melhor pontuação.

2.5.3 struct VotesAVL - votes

Estrutura ordenada por id de post, que guarda a classificação total do dado post, isto é, a difereneça entre o número de votos *UpMod*, e *DownMode* que o cada post obteve.

2.5.4 strcut TagAVL - tags

Estrutura ordenada por id de tag, em que cada nodo é um apontador para uma *struct Tag*, esta estrutura é fundamental para fornecer resposta à interrogação 11.

2.5.5 struct llista - dataPosts

Foi decidido que uma boa forma de obter a eficiencia pretendida, dado que, grande parte das queries pede ordenação cronológica, era guardar a informação numa lista ligada ordenada por datas, em que cada nodo irá corresponder a um período de um mês dos diferentes anos.

Em cada nodo da lista, para além da devida data, que identifica o período, esta vai ter:

- Dois apontadores para duas *struct PostAVL*, em que *postPerguntas* contém apenas os posts que são perguntas, e *postRespostas* os posts que são respostas.
- Dois apontadores para duas *struct LongAVL* (estrutura em que o nodo é apenas um long), em que *usersPerguntas* contém os ids dos users que postaram perguntas nesse período, e *usersRespostas* os ids dos users que postaram respostas nesse período.
- Um apontador para uma *struct TagLL* que contém todas as tags usadas nesse período.

```
typedef struct llista{
    Date data;
    struct PostAVL* postPerguntas;
    struct PostAVL* postRespostas;
```

```
    struct LongAVL* usersPerguntas;
    struct LongAVL* usersRespostas;
    struct TagLL* tags;
    struct llista* next;
    struct llista* prev;
} *LLista;

typedef struct TCD_community{
    struct UserAVL* users;
    struct PostAVL* respostasByParent;
    struct VoteAVL* votes;
    struct TagAVL* tags;
    int usersNumber;
    struct llista* dataPosts;
} *TCD;
```

2.6 Ficheiros arvores.h e linkedlist.h

O encapsulamento de dados é garantida em toda a API, visto que é composto por funçoes autónomas que poderão ser usadas em todo o programa. Além disso,toda a estrutura dos dados encontra-se privada sendo só conhecidas os nomes das funções que usaremos. Para isso criamos uma ficheiro arvores.h e um linkedlist.h que contem todas essas funções.
É de realçar também que sempre que requeremos um resultado,o espaço que esse mesmo resultado quer(e o seu preenchimento) é alocado dentro da própria função.

Iremos apresentar só algumas das funções dos 2 ficheiros para mostrar que o encapsulamento de dados foi cumprido.

2.6.1 User

- *struct UserAVL *newnode(User u)*
Adiciona um novo User á arvore alocando a memória suficiente para a inserção. É de reiterar que as outras funções usam esta funcao para a inserção visto que é nesta que alocamos a memória.

- *struct UserAVL *rightRotate(UserAVL *y)*
Faz a rotação da árvore para a direita

- *struct UserAVL *leftRotate(UserAVL *x)*
Faz a rotação da árvore para a esquerda

- *struct UserAVL *insert(UserAVL *UserAVL,User u)*
Insere um User U na árvore UserAVL usando a funcao *newnode*

- *int getBalance(struct UserAVL *N)*
Verifica se a árvore está balenceada retornando um inteiro

2.6.2 Posts

- *struct PostAVL *avlInit(Post u)*
Inicializa o *Post u* alocando já memória.

-*struct PostAVL *rightRotateP(struct PostAVL *y)*
Faz a rotação da árvore para a direita

-*struct PostAVL *leftRotateP(struct PostAVL *x)*
Faz a rotação da árvore para a esquerda

-*struct PostAVL *insertP(struct PostAVL,Post u,long id)*
Insere o Post u na *struct PostAVL*

-*struct PostAVL *insertParent(struct PostAVL *postAVL*,Post u,long id,UAVL ua)*
Insere o *Post u* ordenado por ParentId.Foi para poder "resolver"a interrogação 10.

*-int getBalanceP(struct PostAVL *N)*
Verifica se a árvore está balanceada retornando o inteiro

-float calculaMedia(Post resposta,UAVL ua
Calcula a média da votação da resposta

2.6.3 Tag

*- struct TagAVL *newNodeT(Tag u,long id)*
Aloca já memória para a inserção da Tag u na *struct TagAVL**

*- struct TagAVL *rightRotateT(struct TagAVL *y)*
Faz a rotação da árvore para a direita

*- struct PostAVL *leftRotateT(struct TagAVL *x)*
Faz a rotação da árvore para a esquerda

*- struct TagAVL *insertT(struct TagAVL *TagAVL,Tag u,long id)*
Insere a Tag u na *struct TagAVL* usando a funcao newNodeT

int alturaT(struct TagAVL *N);
Devolve a altura da árvore

2.6.4 Votes

*- struct VoteAVL *rightRotateV(struct VoteAVL *y)*
Faz a rotação da árvore para a direita

*- struct VoteAVL *leftRotateV(struct VoteAVL *x)*
Faz a rotação da árvore para a esquerda

*- struct VoteAVL *insertV(struct VoteAVL *VoteAVL,Vote u)*
Insere um Vote U na á*struct VoteAVL*

*- int alturaV(struct VoteAVL *N*
Retorna a altura da árvore

*-int getBalanceV(struct VoteAVL *N)*
Verifica atraves de um int se a árvore é balanceada

2.6.5 LongAVL

*-struct LongAVL *newNodeL(long u)*
Cria um novo nodo,alocando já memória suficiente.

*- struct LongAVL *insertL(struct LongAVL *LongAVL,long id)*
Insere um long id na *LongAVL* usando a função newNodeL.

2.6.6 llista

-struct llista insertAtDate(struct llista *ll,Date data,struct Post* post,struct TagLL* tags)*
Insere um *post* e *tags* na lista ordenada por data.

2.6.7 TagLL

*- struct TagLL *tagInit(struct TagLL *tags,char* tag)*
Inicializa o nodo de da *struct TagLL*

2.6.8 UserLL

- struct UserLL *userInit(User u)*
Inicializa a *struct UserLL**

2.7 Parser

É neste modulo que é efetuado o parsing dos ficheiros xml, para tal, são utilizadas as seguintes funções da libraria *libxml*:

- xmlParseFile - efetua o parser do ficheiro xml e devolve um xmlDocPtr
- xmlDocGetRootElement - devolve a raiz do documento, *xmlFreeDoc*. Este valor é comparado com as tags presentes no documento, permitindo assim distinguir os ficheiros fornecidos. Após isto, é processado o conteúdo das "row", acedendo ao campo *xmlChildrenNode* do *xmlFreeDoc* anterior.
- xmlGetProp - para ir buscar o conteúdo de um dado atributo, passado como argumento à função

2.8 Interface

Neste modulo foram respondidas às interrogações propostas no enunciado.

2.8.1 Interrogações

init

- Inicializa a estrutura *TAD_community*.

load

- Efetua o carregamento dos ficheiros xml "Users.xml, Votes.xml, Posts.xml, Tags.xml", para a estrutura fornecida, fazendo uma invocação à função "parsing"para cada ficheiro, função esta que faz as devidas alocações de memória necessárias e guarda nas estruturas as informações obtidas.

Interrogação 1 - info_from_post

- Começa por percorrer a lista até encontrar nas avls de posts o id dado como argumento. Quando encontra, caso seja resposta procura a respetiva pergunta e vai procurar na avl de users o respetivo id do "owner"do post. Devolve um *STR_pair* com o titulo do post e o nome do respetivo autor.

Interrogação 2 - top_most_active

- Começa por converter a avl de users num array de users. Depois esse array é ordenado por número de posts ("countPosts") e no final convertido numa *LONG_List*, valor de retorno.

Interrogação 3 - total_posts

- Testa se a data passada como argumento, *begin* é mais antiga que a *end*, caso o seja, vamos percorrendo a lista desde a data inicial até à final, guardando o numero de nodos das avls de posts presentes em cada nodos da lista. No final é devolvido um *LONG_pair* com o valor total de perguntas e respostas, respetivamente.

Interrogação 4 - questions_with_tag

- Começa por percorrer a lista em ordem inversa, e para cada nodo da lista entre as dadas datas, vai verificar se a tag foi usada nesse período percurrendo a avl *tags* da estrutura passada como argumento *com*. Se tiver sido usada, vai percorrer a avl *postPerguntas* e guarda todas as perguntas com essa tag. No fim devolve uma *LONG_LIST* com os ids das perguntas ordenadas inversamente.

Interrogação 5 - get_user_info

- Começa por percorrer a lista ligada ate ao fim para garantir que procura os mais recentes. Criamos um array para guardar os Posts porque pode ser necessario verificar a data e para isso precisamos da informaçao toda do post. Depois vamos percorrer a lista ligada para tras enquanto não tivermos preenchido o array no index 9 e enquanto a lista não for NULL. Quando encontra um post com o id igual insere no array se encontrar algum que seja mais antigo. Por fim procuramos o user pelo ID na arvore de Users e garantimos a ordenaçao do Array de Posts antes de passar os IDs para a LONG_LIST.

Interrogação 6 - most_voted_answers

- Começa por percorrer a lista por ordem inversa, e para todos os nodos dentro do dado período, vai percorrer todas as avls *postRespostas*, e guardar as respostas com mais votos num array. No final é devolvido uma *LONG_LIST* com os ids das respostas com mais votos ordenada por ordem decrescente do número de votos.

Interrogação 7 - most_voted_answers

- Começa por percorrer a lista por ordem inversa, e para todos os nodos dentro do dado período, vai percorrer todas as avls *postPerguntas*, e guardar as perguntas com mais respostas (valor previamente calculado no parsing) num array. No final é devolvido uma *LONG_LIST* com os ids das perguntas com mais votos ordenada por ordem decrescente do número de respostas.

Interrogação 8 - contains_word

-Começa por percorrer a lista ligada até ao fim e por criar um array de Posts. Enquanto nao tivermos os N Posts e a lista ligada nao for NULL vamos percorrer a arvore de perguntas e verificar se o titulo de cada Post contem a Word desejada. Apos ter o array preenchido ou a lista chegar a NULL, ordenamos o array e passamos para a LONG_LIST os IDs do Posts guardados previamente no array.

Interrogação 9 - both_participated

- Começa por percorrer a lista por ordem inversa, em cada nodo da lista vai procurar se um dos users fez perguntas ou respostas nesse perído. Caso tenha feito perguntas, vai procurar em toda a lista respostas efetuadas pelo outro user passado como argumento, com o *parentId* igual ao *idPost* da pergunta encontrada. Caso tenha feito respostas, quer dizer que o segundo user pode ter feito perguntas ou respostas, e portanto, vai ser feita uma procura de perguntas com o *idPost* igual ao *parentID* da resposta obtida, e caso não encontre, vai então ser feita uma procura por respostas como mesmo *parentID* e em que o *ownerUserId* seja o igual ao id do segundo user passado como argumento. é criada uma *LONG_LIST* ordenada por ordem cronológica inversa, com os ids das perguntas obtidas.

Interrogação 10 - better_answer

- É feita uma procura na avl *respostasByParent* da estrutura *com* passada como argumento à função, pelo id também passado como argumento, e é devolvido o id da melhor resposta. Esta função ficou bem mais simples, devido ao processamento já previamente efetuado no parsing como já foi explicitado.

Interrogação 11 - bestRep

- Começamos por calcular os utilizadores com melhor reputação para o dado intervalo e guardamos num array ordenado por reputação. Depois vamos percorrendo a lista, testando em cada nodo se os utilizadores do array previamente calculado, existem na avl *postPerguntas*. Caso existam procuramos a pergunta correspondente e inserimos as tags respetivas numa avl local do tipo *struct TagAVL* ordenada por ordem alfabética do nome da tag, quando inserirmos uma tag igual o *count* da *struct Tag* é incrementado. Finalmente esta avl é convertida para um array, que é ordenado decrescentemente pelo numero de vezes que a tag foi usada, e guardamos numa *LONG_LIST* todos os ids das tags mais usadas, valor de retorno.

2.9 Conclusão

Acabado o trabalho,cabe-nos fazer uma retrospetiva de todo o trabalho feito. Sentimos dificuldades iniciais para fazer o *parser* e com alguns avanços e recuos conseguimos realizá-lo sem problemas.
Os problemas chegaram depois quando tivemos que criar um estrutura que pudesse atingir os objetivos propostos no enunciado. A nossa maior dúvida foi poder responder as queries com intervalo de tempo e com ajuda dos professores chegamos à conclusão de que deveriamos ter uma lista ligada ordenada por data.
Também consideramos que para poder responder a algumas das interrogações propostas(por exemplo a 10) poderiamos demorar mais tempo a poder fazer o carregamento de toda a estrutura para assim ser mais eficiente as interrogações propostas. Escolhida a estrutura,sentimos dificuldades na realização da interrogação *most_voted_answer* visto que nao guardava os *ids*.
Em suma,todo este projeto foi realizado de modo a responder a todos os problemas propostos no enunciado. Foi respeitado o encapsulamento de modo a proteger os dados,como é apanágio em projetos de grande dimensão como este.