DOCUMENTAÇÃO TP1

Ramon Gonçalves Gonze

INTRODUÇÃO

O objetivo deste trabalho prático é a implementação de conceitos discutidos em sala de aula, como listas encadeadas, pilhas, filas, análise de complexidade de funções, manipulação de vetores, entre outros. Foram utilizados vários TADs (Tipo Abstrado de Dado), que são estruturas nas quais organizam dados e categorias de forma objetiva. O trabalho é uma versão simplificada de uma rede social, como Twitter e Facebook. Para gerar o executável do arquivo, deve-se abrir o prompt de comando (Windows) ou o terminal (Linux), acessar a pasta que contém os 4 arquivos do tp: *main.c*, *funcoes.c*, *funcoes.h* e *Makefile*, e então executar o comando *make*, que compilará corretamente os arquivos através do *Makefile*. Será gerado um arquivo *main.exe* (ou *main.out*). O nome e a extensão do arquivo de entrada a ser testado deverão ser passados por linha de comando no primeiro argumento, como por exemplo: *main entrada01.txt*

IMPLEMENTAÇÃO

❖ Estrutura de dados

Para a implementação do trabalho foram necessárias as criações de 3 TADs:

- TAmigo

Cada usuário possui uma lista de amigos, e esse TAD representa um item nessa lista. Sua composição é:
- int id → Indica o id do usuário que está na lista.
- struct amigo *prox → Um ponteiro para o próximo amigo da lista. Esse ponteiro será *NULL* caso o item em questão seja o último da lista.

- TListaAmigos

Representa a lista de amigos de um usuário. Sua composição é:
- TAmigo *primeiro → Um ponteiro para a cabeça da lista de amigos, e esta cabeça não é um item da lista, ela possui um apontador para o **primeiro** item.
- TAmigo *último → Um ponteiro para o último item da lista de amigos.

- TMensagem

É semelhante ao TAmigo, mas neste caso, esse TAD representa um item em uma lista de mensagens – a timeline. É responsável por armazenar informações de uma mensagem postada na rede social. Sua composição é:
- int id → Armazena o id da mensagem.
- int id_postador → Armazena o id do usuário que postou a mensagem.
- int curtidas → Armazena a quantidade de curtidas que a mensagem possui.
- int *visualizada → Este ponteiro para inteiro será do tamanho da quantidade de usuários que o arquivo de entrada possuir. Cada posição desse vetor indicará um usuário, e poderá

possuir dois valores: 1 – Não deve ser visualizada pelo usuário; 0 – Deve ser visualizada pelo usuário.
- int tempo → Armazena o tempo em que a mensagem foi postada.
- char *mensagem → Armazena o texto da mensagem postada.
- struct mensagem *prox → Um ponteiro para a próxima mensagem da lista. Esse ponteiro será *NULL* caso o item em questão seja o último da lista.

- TTimeline

Representa a lista de mensagens da rede social, como uma linha do tempo. Sua estrutura é de uma pilha de execução, e sua composição é:
- TMensagem *fundo → Um ponteiro para o fundo da pilha. Não é armazenada nenhuma mensagem no *TMensagem* que este ponteiro aponta. A função deste é possuir um apontador para o primeiro item no fundo da lista.
- TMensagem *topo → Um ponteiro que aponta para o topo da pilha de mensagens.

- TUsuario

Esse TAD é utilizado para representar um usuário da rede social, e possui as seguintes variáveis:
- int id → Armazena o id do usuário.
- char *nome → Armazena o nome do usuário.
- TListaAmigos *amigos → Armazena a lista de amigos que o usuário possui.

❖ Funções e procedimentos

Podemos dividir os TADs em 2 categorias: Usuários e Mensagens.

- Usuários

Os TADs que representam os usuários são: *TAmigo*, *TListaAmigos* e *TUsuario*. Eles possuem as seguintes funções:

TListaAmigos *criaListaAmigos()
Essa função cria uma nova lista de amigos. Uma nova variável do tipo *TAmigo* é criada para ser a cabeça da lista, e também é criado uma nova várivavel do tipo *TListaAmigos*, que terá seus ponteiros *primeiro* e *ultimo* apontados para a variável cabeça. Ela retorna o endereço de memória alocado para a nova lista.

void adicionaAmigo(TListaAmigos *lista, int id)
Essa função recebe uma lista de amigos de algum usuário, e então adiciona um novo amigo ao fim desta. Este novo amigo receberá o id passado pelo segundo parâmetro.

TListaAmigos *verAmigos(TUsuario *usuario)
Função que recebe como parâmetro um usuário e retorna a sua lista de amigos.

int idUsuario(TUsuario **usuarios, int num_usuarios, int id)
O objetivo dessa função é retornar o id do usuário que está relacionado à variável **usuários**, que foi passada como primeiro parâmetro. Na execução da função **main**, é criado um vetor do tipo *TUsuario* para alocar todos os usuários do arquivo de entrada. Logo, cada posição do vetor, corresponde à um usuário. Exemplo: Se houver dois usuários, com ids 100 e 200, o vetor usuários terá tamanho 2 e as respectivas posições destes usuários serão 0 e 1.

Id do Usuário	Posição no vetor de usuários na função main
100	0
200	1
300	2

Exemplo de composição do vetor usuários na função main.

void iniciarAmizade(TUsuario **usuarios, int num_usuarios, int id1, int id2)
Essa função inicia a amizade entre os usuários indicados por *id1* e *id2*. Ela insere o usuário *id1* no fim da lista de amigos do usuário *id2* e vice-versa.

void cancelarAmizade(TUsuario **usuarios, int num_usuarios, int id1, int id2)
Essa função cancela a amizade entre os usuários indicados por *id1* e *id2*. Ela remove o usuário *id2* da lista de amigos do usuário *id1* e vice-versa.

- Mensagens

Os TADs que representam as mensagens são: *TMensagem* e *TTimeline*. Eles possuem as seguintes funções:

TTimeline* criaTimeline()
Essa função cria uma nova pilha, que será a timeline da rede social. Uma nova variável do tipo *TMensagem* é criada para ser o fundo da pilha. Também é criada uma nova variável do tipo TTimeline. Os seus dois ponteiros, *fundo* e *topo*, apontarão para o fundo, que acabou de ser criado. Por fim, ela retorna o endereço apontado pela nova pilha.

void postarMensagem(TUsuario **usuarios, TTimeline *timeline, int num_usuarios, int id_usuario, int id_mensagem, char *msg, int tempo)
Essa função faz a postagem de uma nova mensagem na timeline da rede social. As informações dessa nova mensagem serão preenchidas de acordo com os parâmetros passados. Para selecionar qual usuário deverá ver a nova mensagem, o seguinte critério é avaliado: no momento da postagem, se o usuário for amigo do usuário que postou a mensagem (identificado pelo quarto parâmetro *id_usuario*), ele poderá lê-la na timeline, e o vetor *visualizada*, na posição do usuário analisado, será preenchido com o valor 0, caso contrário será preenchido com o valor 1. Após a mensagem estar corretamente preenchida, é feita a sua inserção no topo da pilha de mensagens, ou seja, da timeline.

void exibirTimeline(TUsuario **usuarios, TUsuario *usuario, TTimeline *timeline, FILE **arq_saida, int num_usuarios)
Essa função tem o objetivo de gerar um arquivo de log, para cada arquivo de entrada. Quando um usuário na rede social exibe a timeline, a função exibirá as mensagens da pilha. A mensagem do topo será a primeira a ser exibida, depois a de baixo, e assim consecutivamente até o fundo da pilha. Percebe-se que todas as mensagens da pilha serão analisadas, mas só serão escritas no arquivo de log aquelas que possuírem o vetor *visualizada,* na posição do usuário que está exibindo a timeline, com valor 0.

void curtirMensagem(TUsuario **usuarios, TTimeline *timeline, int id_usuario, int id_mensagem, int num_usuarios)
Essa função tem objetivo de reorganizar a timeline da rede social de acordo com a popularidade das mensagens, que são as curtidas que cada uma possui. O primeiro passo é incrementar em +1 a variável *curtidas* da mensagem. A mensagem é encontrada na pilha pelo parâmetro *id_mensagem*. O segundo passo é a reorganização da timeline. Uma nova curtida, faz com que a mensagem deva ser colocada no início da timeline. A função retira a mensagem de seu lugar, e a coloca no topo da pilha.

❖ Programa principal

A função do programa principal neste trabalho é ler um arquivo de entrada, seus respectivos usuários e ações executadas na rede social, para então gerar um arquivo .txt de log. O nome do arquivo de entrada é passado por linha de comando no momento da execução do programa. Se o arquivo descrito existir, o restante do programa é executado, caso contrário é exibida uma mensagem de erro e o programa finaliza.

Sendo aberto um arquivo, a função lê linha por linha do arquivo, armazenando os usuários da rede social, e as ações que estes executaram. Para cada ação em um tempo, a função responsável por aquela ação é chamada.

❖ Análise de complexidade

Para a análise de complexidade, iremos usar como parâmetro a quantidade de mensagens postadas, que será representado por ***n***.

Função criaListaAmigos: Cria duas variáveis, faz alguns comandos de atribuição e retorna o conteúdo da variável do tipo *TListaAmigos* que foi criada. Complexidade**: O(1)**.

Função criaTimeline: Cria duas variáveis, faz alguns comandos de atribuição e retorna o conteúdo da variável do tipo *TTimeline* que foi criada. Complexidade**: O(1)**.

Função adicionaAmigo: Cria uma nova celula da lista de amigos, e adiciona esta no final da lista. Complexidade**: O(1)**.

Função verAmigos: Recebe como parâmetro uma variável do tipo *TUsuario* e retorna a sua lista de amigos. Complexidade**: O(1)**.

Função idUsuario: A função possui um laço de repetição que vai de 0 até *num_usuarios – 1*. A cada interação com o laço, é feito uma comparação. Ela não depende da quantidade de mensagens postadas, logo sua complexidade é **O(1)**.

Função postarMensagem: A função executa algumas atribuições antes do primeiro laço: for(i = 0; i < num_usuarios; i++). Este laço é executado *num_usuarios* vezes. Dentro deste, existe outro laço que percorre a lista de usuários do usuário. O tamanho máximo dessa lista de usuários é *num_usuario – 1*, que é quando o usuário é amigo de todas as pessoas da rede. A função não depende da quantidade de mensagens postadas, logo sua complexidade é **O(1)**.

Função iniciarAmizade: Esta função 2 laços de repetições. O primeiro laço *while* percorre a lista de amigos de usuário, para verificar se este já é amigo da pessoa. O caso esperado é que não encontre ela na lista, então ele irá percorrer a lista até o fim. No pior caso, a pessoa será amiga de todos os usuários da rede, e a lista, portanto, terá tamanho *num_usuarios – 1*. O segundo laço faz exatamente a mesma coisa que o primeiro, só que agora para o segundo usuário. Ambos os laços possuem execuções O(1) dentro deles. A função, no geral, não depende da quantidade de mensagens postadas, logo sua complexidade é **O(1)**.

Função cancelarAmizade: A função possui dois laços de repetições, na qual cada um percorre a lista de amigos de um usuário. O pior caso acontece quando estes dois usuários são amigos de todas as pessoas da rede social, logo suas listas de amigos terão *num_usuarios – 1* elementos. Quando o usuário a ter sua amizade cancelada é o último da lista de amigos do outro, o laço se repetirá *num_usuarios – 1* vezes. Como a função não depende da quantidade de mensagens postadas, sua complexidade é **O(1)**.

Função exibirTimeline: A função possui dois laços de repetições. Os laços percorrem a pilha de mensagens, a timeline, do topo até o fundo. Logo, a complexidade desses laços dependerá da quantidade de mensagens existentes na timeline. Considerando *n = número de mensagens*, temos que o laço externo percorrerá de *n* até *1*, decrescendo 1 por vez, Logo será executado *n* vezes. Já o laço interno percorrerá de *1* até *n - i*, sendo *i* a *i-ésima* vez que o laço externo está sendo percorrido. Por fim, o laço interno possui somente operações O(1). Analisando:

$$\sum_{i=1}^{n}\left(\sum_{j=1}^{n-i} 1\right) = \sum_{i=1}^{n} n - i = \sum_{i=1}^{n} n - \sum_{i=1}^{n} i =$$

$$n^2 - \frac{n(n+1)}{2} = n^2 - \frac{n^2+n}{2} = \frac{n^2}{2} - \frac{n}{2}$$

Logo, sua complexidade é de **$O(n^2)$**.

Função curtirMensagem: Nesta função há dois laços de repetição intercalados. O laço externo, for(i = 0; i < num_usuarios; i++), é percorrido *num_usuarios* vezes. O laço interno percorre a lista de amigos do usuário analisado. No pior caso, este usuário será amigos de todos da rede social, logo sua lista terá *num_usuarios - 1* elementos. O laço interno possui uma comparação e uma atribuição somente. Como a função não depende da quantidade de mensagens postadas, sua complexidade é **O(1).**

Função main: Há três laços de repetição na função:

for(i = 0; i < num_usuarios; i++)
O laço é percorrido de 0 até *num_usuarios - 1*, e possui um laço interno, o while(amigo != NULL).

> **while(amigo != NULL)**
> Este laço adiciona na lista de amigos de um usuário todos aqueles que ele já possui amizade. No pior caso, o usuário será amigo de todos os outros, então sua lista de amigos terá *num_usuarios - 1* elementos. Dentro deste laço há uma atribuição e a chamada da função *adicionarAmigo*.

while(!feof(arq_entrada))
O laço é executado até que se chegue ao final do arquivo de entrada que está sendo lido. Logo, ele será executado *qtd_acoes* vezes, inclusive a ação "postar mensagem". Cada execução desse laço corresponde à leitura de uma ação que está no arquivo. Há um *switch* que seleciona qual função deverá ser chamada para executar a ação solicitada, e cada caso do switch faz algumas operações O(1).

Analisando a complexidade final, e considerando os laços descritos acima, respectivamente como laço 1, 2 e 3:

Laço 1: O(1)
    Laço 2: O(1)
Laço 3: O(n)

Complexidade final com base no *número de mensagens postadas*: **O(n)**

CONCLUSÃO

O trabalho foi feito baseado no conteúdo discutido em sala e nos slides fornecidos pelo professor. A aplicação de técnicas e métodos foram utilizadas sem grandes problemas. A utilização de listas e pilhas se mostrou importante para a organização do código e manipulação dos elementos de uma rede social. O algoritmo conseguiu atender o objetivo proposto pelo enunciado.

BIBLIOGRAFIA

CPLUSPLUS. **function sprintf**. Disponível em: <http://www.cplusplus.com/reference/cstdio/sprintf/>. Acesso em: 01 nov. 2016.