EXERCÍCIOS – LISTAS, FILAS E PILHAS

INSTRUÇÃO GERAL: para cada estudo de caso, gere uma pasta separada com cada versão solicitada.

DICA: Nos exercícios onde pede-se a implementação da interface do zero, ou mesmo a adição de novas features, faça em etapas. Implemente um programa que vá utilizando as features disponibilizadas na interface e na medida em que implementar uma nova feature, faça uso dela no programa. Vá debugando para garantir que cada nova feature implementada tenha sido implementada corretamente.

1. Faça uma adaptação da interface da lista com encadeamento simples apresentada em sala de aula. A nova versão da lista deve armazenar, de informação, um número em ponto flutuante, com dupla precisão (*double*). A partir dessa nova versão, crie um programa que disponibiliza um menu de opções para o usuário: opção 0 para inserir novo valor, opção 1 para remover um valor da lista, opção 2 para exibir a lista atual e opção 3 para encerrar o programa. Ao encerrar o programa, você deve liberar os elementos da lista, que foram alocados dinamicamente.

2. Faça uma melhoria da interface da lista com encadeamento simples apresentada em sala de aula. Pede-se uma nova versão com as seguintes features:

   a) Uma função que receba um ponteiro para a head da lista e que tenha como valor de retorno o comprimento da lista encadeada, (isto é, que calcule o número de nós de uma lista). Essa função deve ter o protótipo:
   ```
   int lst_comprimento (ListaInt* l);
   ```
   b) Uma função que retorna o número de nós da lista que possuem o campo de valor maior do que x. Essa função deve obedecer ao protótipo:
   ```
   int lst_maiores(ListaInt *l, int x);
   ```
   c) Uma função que retorne o último valor da lista encadeada de inteiros. Essa função deve obedecer ao protótipo:
   ```
   int lst_ultimo (ListaInt *l)
   ```
   d) Uma função que realiza a concatenação de duas listas encadeadas, transferindo para o final da primeira lista os elementos da segunda. A primeira lista da função será a lista resultado e a função deve ter o protótipo:
   ```
   void lst_concatena(ListaInt *l1, ListaInt *l2);
   ```
   e) Uma função que insere um elemento no fim da lista. A função deverá ter o protótipo:
   ```
   void lst_insere_fim(ListaInt *l, int v);
   ```

   Agora faça um programa que utilize unicamente as funções da interface da lista de inteiros. Crie um programa que:
   - Solicita ao usuário quantos elementos ele quer digitar na primeira lista;
   - Faça a leitura dos valores e vá inserindo os novos elementos sempre no início da lista;

- Solicita quantos elementos o usuário quer digitar na segunda lista;
- Faça a leitura dos valores digitados pelo usuário e insira os novos elementos sempre no fim da lista;
- Exiba o conteúdo de ambas as listas digitadas;
- Exiba o comprimento das duas listas;
- Exiba o último valor digitado em cada uma das listas;
- Solicite ao usuário para digitar um valor inteiro e mostre quantos elementos em cada uma das listas, tem valores superiores ao valor digitado pelo usuário;
- Por fim, concatene as duas listas e exiba a lista concatenada;
- Libere a memória utilizada por ambas as listas.

3. Faça uma adaptação da interface da lista com encadeamento simples apresentada em sala de aula. Altere a interface da lista com encadeamento simples para uma versão de lista com encadeamento duplo. Além disso, inclua uma funcionalidade para imprimir a lista de forma reversa:

   void lst_imprime_reverso(ListaInt *l)

   Essa função deve imprimir a lista começando pelo último elemento até o primeiro elemento. Para mostrar o uso da nova versão da lista, crie um código que importa essa interface e apresenta um menu de opções para o usuário: 0 – inserir novo elemento; 1 – remover elemento; 2 – imprimir a lista; 3 – imprimir a lista de forma reversa; 4 – sair. Ao sair do programa, libere a lista da memória.

4. Faça uma adaptação da interface da lista com encadeamento simples apresentada em sala de aula. Altere a interface da lista com encadeamento simples para uma versão de lista com encadeamento simples circular. Faça as modificações necessárias para que essa lista funcione. Além disso, nesse caso, a função de inserção tem que inserir novos elementos no final da lista.

   Para mostrar o uso da nova versão da lista, crie um código que importa essa interface e apresenta um menu de opções para o usuário: 0 – inserir novo elemento; 1 – remover elemento; 2 – imprimir a lista; 3 – sair. Ao sair do programa, libere a lista da memória.

   DICA: A head da lista é uma estrutura útil para guardar metadados a respeito da lista. Informações que possam ser relevantes sobre o estado atual da lista, ou mesmo informações que facilitam a busca de certas características da lista. Por exemplo, o metadado mais básico é o início da lista. Em listas circulares, o último elemento tem que apontar para o primeiro elemento. Uma forma de facilitar a abordagem é guardar, além do primeiro elemento, o último elemento da lista na estrutura que representa a lista. Isso também facilita para o caso onde a inserção da lista é no final, como o exercício pede. Todavia, isso vai requerer que, sempre que a lista for alterada, os metadados da head sejam atualizados de forma concomitante.

5. Aplique a ideia de guardar o último elemento da lista no head da lista, para uma lista duplamente encadeada não-circular. Utilize a implementação resultante do exercício

3, que foi construída com o código disponibilizado para a interface de uma lista simplesmente encadeada não-circular. Essa lista tem que inserir elementos ao fim da lista. Faça as alterações necessárias para utilizar esse metadado do head da lista duplamente encadeada não circular. Para mostrar o uso da nova versão da lista, crie um código que importa essa interface e apresenta um menu de opções para o usuário: 0 – inserir novo elemento; 1 – remover elemento; 2 – imprimir a lista; 3 – imprimir a lista de forma reversa; 4 – sair. Ao sair do programa, libere a lista da memória.

6. Adapte a interface da lista simplesmente encadeada, disponibilizada, para guardar informações de alunos. A informação é uma struct de alunos definida conforme a descrição abaixo:

```
typedef struct aluno Aluno;
struct aluno {
char nome[81]; // nome
float nota;
};
```

A interface dessa lista deve ser a seguinte:

```c
#ifndef LISTA_ALUNO_H_
#define LISTA_ALUNO_H_

typedef struct aluno Aluno;
typedef struct lista_aluno_no ListaAlunoNo;
typedef struct lista_aluno ListaAluno;

// Interface das funções para manipular aluno
Aluno* cria_aluno(char* nome, float nota);
void libera_aluno(Aluno* a);
void exibe_aluno(Aluno* a);

// Interface das funções da lista
ListaAluno* lst_cria(void);
void lst_insere(ListaAluno *l, Aluno* a);
void lst_imprime(ListaAluno *l);
int lst_pertence(ListaAluno *l, Aluno* a);
void lst_retira(ListaAluno *l, Aluno* a);
int lst_vazia(ListaAluno *l);
void lst_libera(ListaAluno *l);
Aluno* lst_maior_nota(ListaAluno *l);
Aluno* lst_menor_nota(ListaAluno *l);
```

```
Aluno* lst_retorna_aluno(ListaAluno *l, char*
nome);
float lst_media_alunos(ListaAluno *l);

#endif
```

Implemente um programa que faz uso dessa interface. O programa deve apresentar um menu ao usuário permitindo as seguintes funcionalidades:

1. Cadastrar novo aluno;
2. Remover aluno pelo nome;
3. Buscar o nome e a nota do aluno com maior nota;
4. Buscar o nome e a nota do aluno com menor nota;
5. Calcular a média das notas dos alunos;
6. Exibir os nomes e as notas de todos os alunos;
7. Sair do programa.

Ao sair do programa, a memória deve ser liberada. O head da lista, obrigatoriamente deve ter informação a respeito do tamanho atual da lista (portanto, atualizações na lista irão requerer atualizações no head). Isso facilitará a implementação da função lst_comprimento, poupando o gasto computacional de ter que ficar atravessando múltiplas vezes a lista para contar quantos elementos tem na lista. Conforme já incitado anteriormente, o head da lista existe para guardar os metadados da lista como um todo e pode ser utilizado para poupar gasto computacional com relação a certas informações. Essa é uma escolha de design de quem implementa a interface, usualmente visando poupar gasto computacional em certas circunstâncias.