Estrutura de Dados 2

Envio: a atividade deve ser enviada para o repositório do aluno dentro da pasta **Lista 2 - Listas, Pilhas, Filas**. Cada questão deve ficar em uma subpasta própria (ex.: questao-1/, questao-2/, questao-3/), contendo código-fonte (.c/.h), e README.md com instruções de compilação/execução e exemplos.

Questão 1 — Lista ligada para Controle de Estoque

Uma loja mantém um catálogo de produtos com entradas, saídas e ajustes ao longo do dia. Implemente uma lista simplesmente encadeada para gerenciar o estoque. Considere a seguinte estrutura para Produto (você pode realizar modificações somente para adicionar novos atributos).

```c
typedef struct Produto {
  char codigo[16];
  char nome[64];
  int qtd;
  float preco;
  struct Produto *prox;
} Produto;
```

Operações (todas com malloc/free):

1. Produto* inserir_inicio(Produto *L, Produto p);
2. Produto* inserir_ordenado_por_codigo(Produto *L, Produto p); (se o código já existir, atualizar qtd e preco)
3. Produto* remover_por_codigo(Produto *L, const char *codigo);
4. Produto* ajustar_quantidade(Produto *L, const char *codigo, int delta_qtd); (não permitir qtd negativa)
5. Produto* buscar(Produto *L, const char *codigo);
6. void listar(Produto *L);
7. Produto* limpar(Produto *L);

Aplicação (em main.c): menu textual com comandos:

```
ADD <cod> "<nome>" <qtd> <preco>
UPD <cod> <delta_qtd>
DEL <cod>
FIND <cod>
LIST
REL
EXIT
```

REL mostra: total de itens distintos, valor total do estoque (Σ qtd*preco) e o produto com maior valor imobilizado.

Toda a interação é somente pelo terminal.

Documente no README a complexidade Big-O de cada operação.

Questão 2 — Fila para Venda Digital de Ingressos

Uma plataforma abre vendas às 10h e cria uma única fila de espera de clientes. Ao iniciar a venda, o sistema processa pedidos em ordem FIFO enquanto houver ingressos. Considere as seguintes estruturas para Cliente e para a fila:

```
typedef struct Cliente {
  char id[24];
  int qtd;
  struct Cliente *prox;
} Cliente;

typedef struct {
  Cliente *ini, *fim;
} Fila;
```

Operações:

1. void inicializar(Fila*);
2. void enfileirar(Fila*, Cliente); // enqueue
3. bool desenfileirar(Fila*, Cliente *out); // dequeue
4. bool vazia(Fila*);
5. void limpar(Fila*);

Aplicação (em main.c):

Parâmetro inicial: estoque_inicial informado pelo usuário no começo da execução.

```
ADD <id> <qtd>
START
REL
EXIT
```

ADD insere cliente no final da fila.

START atende clientes sequencialmente:

Se qtd solicitada ≤ estoque, confirma e subtrai do estoque.

Se qtd > estoque e ainda houver algum ingresso, ofereça a quantidade disponível e peça confirmação por entrada do usuário (S/N); se aceitar, vende; se não, registra como não atendido.

Se estoque = 0, para o processamento. E exibe uma mensagem de finalização.

REL exibe: total vendido, número de clientes atendidos, top-3 maiores compras, quantidade restante em estoque e lista de não atendidos.

Questão 3 — Pilha de Processos (alocação dinâmica) — versão simplificada

Simule o stack de um processo do SO, contendo apenas chamadas e retornos de funções.

```c
typedef struct Frame {
  char func[48];
  int sp;               // stack pointer lógico/simulado
  struct Frame *baixo; // próximo elemento em direção ao fundo
} Frame;

typedef struct {
  Frame *topo;
} Pilha;
```

Operações obrigatórias:

1. void pilha_init(Pilha*);
2. bool push(Pilha*, const char *func, int sp);
3. bool pop(Pilha*, Frame *out); // retorna false se vazia
4. bool peek(Pilha*, Frame *out);
5. bool vazia(Pilha*);
6. void limpar(Pilha*);

Aplicação (em main.c): comandos simples:

```
CALL <nomeFunc> <sp>
RET
PEEK
TRACE
EXIT
```

CALL empilha novo frame.

RET desempilha; se vazia, mostrar mensagem de erro amigável.

PEEK mostra o frame do topo sem removê-lo.

TRACE imprime a pilha do topo ao fundo, numerando níveis.