E-fólio Global | Folha de resolução do E-fólio

**UNIDADE CURRICULAR:** Laboratório de Programação

**CÓDIGO:** 21178

**DOCENTE:** Vítor Rocio / José Póvoa

**A preencher pelo estudante**

**NOME:** Andreia Isabel Teófilo Agostinho Romão

**N.º DE ESTUDANTE:** 1702430

**CURSO:** Licenciatura Engenharia Informática

**DATA DE ENTREGA:** 20 Junho 2022

## TRABALHO / RESOLUÇÃO:

## A) código do programa em C;

## main.c

```c
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <locale.h>

#include "ler_escrever.h"
#include "distribuicao.h"

int main() {

	setlocale(LC_ALL, "Portuguese");

	TSListaMesas* mesas = NULL;
	TSListaClientes* clientes = NULL;

	mesas = carregarFicheiroMesas(mesas);
	clientes = carregarFicheiroClientes(clientes);

	// testes de controlo do carregamento das listas
	imprimeListaMesas(mesas);
	imprimeListaCliente(clientes);

	// aloca os grupos às mesas
	atribuiMesaGrupo(clientes, mesas);
	// imprime o reultado da alocação dos grupos às mesas
	imprimeListaDistribuicao(clientes);

	// libertação da memoria das listas
	libertarMemoriaMesas(mesas);
	libertarMemoriaClientes(clientes);
}
```

## ler_escrever.h

```c
#ifndef LER_ESCREVER_H
#define LER_ESCREVER_H
// Prototipo das funções

// ficheiro com o cadastro das mesas
#define FICHEIRO_MESAS "mesas.csv"

// ficheiro com o cadastro dos clientes
#define FICHEIRO_CLIENTES "clientes.csv"

// mesa vazia
#define MESA_VAZIA 0

// topologias das mesas
#define MESA_2 2
#define MESA_4 4
#define MESA_6 6
#define MESA_8 8
```

```c
// mesas
typedef struct SMesas {
    int idMesa;
    int nLugares;
    int nOcupados;
} TSMesas;

// define a estrutura da lista dos alunos com um apontador para a estrutura
produto
typedef struct SListaMesas {
    TSMesas  mesa;
    struct SListaMesas* seg;
}TSListaMesas;

// clientes
typedef struct SClientes {
    char grupo ;
    int pessoas;
    int mesaAtribuida;
} TSClientes;

// define a estrutura da lista dos clienstes com um apontador para a estrutura
clientes
typedef struct SListaClientes {
    TSClientes  cliente;
    struct SListaClientes* seg;
}TSListaClientes;

// carrega ficheiro das mesas para a estrutura
TSListaMesas* carregarFicheiroMesas(TSListaMesas* lista);

// imprime a lista das mesas
void imprimeListaMesas(TSListaMesas* lista);

// carrega ficheiro dos Grupos de clietes para a estrutura
TSListaClientes* carregarFicheiroClientes(TSListaClientes* lista);

// imprime a lista dos grupos
void imprimeListaCliente(TSListaClientes* lista);

// imprime a lista final com as distribuições dos grupos pelas mesas
void imprimeListaDistribuicao(TSListaClientes* clientes);
#endif
//Fim de ler_escrever.h
```

## ler_escrever.c

```c
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <locale.h>

#include "ler_escrever.h"
#include "distribuicao.h"

/*
* carregar dados dos ficheiro mesas para a lista mesas
*/
TSListaMesas* carregarFicheiroMesas(TSListaMesas* lista) {
```

```
    FILE* fileOne = fopen(FICHEIRO_MESAS, "r");
    char bufferTemp[500] = { '\0' };

    int tam = 0;

    TSMesas mesasTemp;

    char* token;

    if (fileOne == NULL) {
        //printf("\n\t\t\tErro ao abrir ficheiro !\n");
        return lista;
    }

    if (lista != NULL) {
        printf("\n\t\t\t Lista já preenchida !\n");
        return lista;
    }

    while (!feof(fileOne)) {

        if (fgets(bufferTemp, 500, fileOne) != NULL) {

            token = strtok(bufferTemp, ",");

            if (token != NULL) {
                mesasTemp.idMesa = atoi(token);
                token = strtok(NULL, ",");
                if (token != NULL) {
                    mesasTemp.nLugares = atoi(token);
                    token = strtok(NULL, "\n");
                    if (token != NULL) {
                        tam = strlen(token) - 1;
                        if (token[tam] == '\n') {
                            token[tam] = '\0';
                        }
                        mesasTemp.nOcupados = atoi(token);
                    }
                }

            }
        }
        // inserir os dados na lista
        lista = insereOrdenadoMesa(lista, &mesasTemp);
    }
    return lista;
    fclose(fileOne);
}

/*
* imprime a lista mesas (testes)
*/
void imprimeListaMesas(TSListaMesas* lista) {
    TSListaMesas* atual = lista;

    if (atual == NULL) {
        printf("\n\n##--- Lista vazia ---##\n\n");
        return;
    }
    else {
        while (atual != NULL) {
```

```
            printf("\n\t\t%d,%d,%d", atual->mesa.idMesa, atual->mesa.nLugares,
atual->mesa.nOcupados);
            atual = atual->seg;
        }
    }
}

/*
* carregar dados dos ficheiro clientes para a lista clientes
*/
TSListaClientes* carregarFicheiroClientes(TSListaClientes* lista) {

    FILE* fileOne = fopen(FICHEIRO_CLIENTES, "r");
    char bufferTemp[500] = { '\0' };

    int tam = 0;

    TSClientes clientesTemp;

    char* token;

    if (fileOne == NULL) {
        //printf("\n\t\t\tErro ao abrir ficheiro !\n");
        return lista;
    }

    if (lista != NULL) {
        printf("\n\t\t\t Lista já preenchida !\n");
        return lista;
    }

    while (!feof(fileOne)) {

        if (fgets(bufferTemp, 500, fileOne) != NULL) {

            token = strtok(bufferTemp, ",");

            if (token != NULL) {
                clientesTemp.grupo = token[0];
                token = strtok(NULL, "\n");
                if (token != NULL) {
                    tam = strlen(token) - 1;
                    if (token[tam] == '\n') {
                        token[tam] = '\0';
                    }
                    clientesTemp.pessoas = atoi(token);
                }
            }
        }
        // inserir os dados na lista
        lista = insereOrdenadoClientes(lista, &clientesTemp);
    }
    return lista;
    fclose(fileOne);
}

/*
* imprime a lista clientes (testes)
*/
void imprimeListaCliente(TSListaClientes* lista) {
    TSListaClientes* atual = lista;
```

```
    if (atual == NULL) {
        printf("\n\n##--- Lista vazia ---##\n\n");
        return;
    }
    else {
        while (atual != NULL) {
            printf("\n\t\t%c,%d,%c", atual->cliente.grupo, atual-
>cliente.pessoas, atual->cliente.mesaAtribuida);
            atual = atual->seg;
        }
    }
}

/*
* imprime a lista da distribuição
*/
void imprimeListaDistribuicao(TSListaClientes* clientes) {
    TSListaClientes* atual = clientes;

    if (atual == NULL) {
        printf("\n\n##--- Lista vazia ---##\n\n");
        return;
    }
    else {
        while (atual != NULL) {
            if (atual->cliente.mesaAtribuida == MESA_VAZIA) {
                printf("\nGrupo %c - aguarda", atual->cliente.grupo);
            }
            else {
                printf("\nGrupo %c - mesa %d", atual->cliente.grupo, atual-
>cliente.mesaAtribuida);
            }
            atual = atual->seg;
        }
    }
}
```

## distribuicao.h

```
#ifndef DISTRIBUICAO_H
#define DISTRIBUICAO_H
// Prototipo das funções

// insere os dados ordenados nas listas
TSListaMesas* insereOrdenadoMesa(TSListaMesas* ordenada, TSMesas* novoMesaa);
TSListaClientes* insereOrdenadoClientes(TSListaClientes* ordenada, TSClientes*
novoCliente);

//LIbertação da memória alocada às listas
TSListaMesas* libertarMemoriaMesas(TSListaMesas* lista);
TSListaClientes* libertarMemoriaClientes(TSListaClientes* lista);

// atribuição dos grupos às mesas disponiveis
void atribuiMesaGrupo(TSListaClientes* clientes, TSListaMesas* mesas);
#endif
//Fim de distribuicao.h
```

## distribuicao.c

```c
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <locale.h>

#include "ler_escrever.h"
#include "distribuicao.h"

/*
* adiciona os valores das mesas de forma ordenados por IDMesa
*/
TSListaMesas* insereOrdenadoMesa(TSListaMesas* ordenada, TSMesas* novoMesa) {

	TSListaMesas* antes;
	TSListaMesas* atual;

	TSListaMesas* elemento = (TSListaMesas*)malloc(sizeof(TSListaMesas));

	if (elemento == NULL) {
		printf("Erro de alocacao\n");
		return 0;
	}
	else {
		// atribuir os valores ao elemento
		elemento->mesa.idMesa = novoMesa->idMesa;
		elemento->mesa.nLugares = novoMesa->nLugares;
		elemento->mesa.nOcupados = novoMesa->nOcupados;

		elemento->seg = NULL;

		atual = ordenada;
		antes = ordenada;

		while (atual != NULL && atual->mesa.idMesa < novoMesa->idMesa) {
			antes = atual;
			atual = atual->seg;
		}

		if (antes == atual) {
			// insere elemento no início //
			elemento->seg = atual;
			return elemento;
		}

		else {
			// insere elemento no meio da lista //
			elemento->seg = antes->seg;
			antes->seg = elemento;
		}
	}
		return ordenada;
}

/*
* adiciona os valores dos Grupos de forma ordenados por grupo
*/
TSListaClientes* insereOrdenadoClientes(TSListaClientes* ordenada, TSClientes*
novoCliente) {
```

```
	TSListaClientes* antes;
	TSListaClientes* atual;

	TSListaClientes* elemento =
(TSListaClientes*)malloc(sizeof(TSListaClientes));

	if (elemento == NULL) {
		printf("Erro de alocacao\n");
		return 0;
	}
	else {
		// atribuir os valores ao elemento
		elemento->cliente.grupo = novoCliente->grupo;
		elemento->cliente.pessoas = novoCliente->pessoas;
		elemento->cliente.mesaAtribuida = MESA_VAZIA;

		elemento->seg = NULL;

		atual = ordenada;
		antes = ordenada;

		while (atual != NULL && atual->cliente.grupo < novoCliente->grupo)
{
			antes = atual;
			atual = atual->seg;
		}

		if (antes == atual) {
			// insere elemento no início //
			elemento->seg = atual;
			return elemento;
		}

		else {
			// insere elemento no meio da lista //
			elemento->seg = antes->seg;
			antes->seg = elemento;
		}
	}

	return ordenada;
}

/*
*libertar a memoria de toda a lista das mesas
*/
TSListaMesas* libertarMemoriaMesas(TSListaMesas* lista) {
	TSListaMesas* aux;
	while (lista != NULL) {
		aux = lista->seg;
		free(lista);	//libertar a lista
		lista = aux;
	}
	return lista;// retorna a lista vazia
}

/*
*libertar a memoria de toda a lista dos clientes
*/
TSListaClientes* libertarMemoriaClientes(TSListaClientes* lista) {
	TSListaClientes* aux;
	while (lista != NULL) {
```

```
                aux = lista->seg;
                free(lista);    //libertar a lista
                lista = aux;
        }
        return lista;// retorna a lista vazia
}

/*
* devolver a tipologia da mesa de acordo com o numero de pessoas do grupo a
alocar
*/
int topologiaCliente(int pessoas) {

        if (pessoas <= MESA_2) {
                return MESA_2;
        }
        else if (pessoas <= MESA_4) {
                return MESA_4;
        }
        else if (pessoas <= MESA_6) {
                return MESA_6;
        }
        else if (pessoas <= MESA_8) {
                return MESA_8;
        }
        else {
                return MESA_VAZIA;
        }
}

/*
* Senta um grupo na mesa com a tipologia correta
* - Mesa tem de estar desocupada
* - Ter numero de lugares suficientes para o tamanho do grupo
* - Ser da Tipologia correta
*/
int sentarGrupo(TSListaMesas* mesas, int topologia, int pessoas) {
        TSListaMesas* atualMesas = mesas;

        if (atualMesas == NULL) {
                printf("\n\n##--- Lista vazia ---##\n\n");
                return MESA_VAZIA;
        }
        else {
                while (atualMesas != NULL) {// mesa livre e topologia certa para o
grupo a sentar
                        if (atualMesas->mesa.nOcupados ==0 && atualMesas-
>mesa.nLugares >= pessoas && atualMesas->mesa.nLugares <= topologia) {
                                // sentar pessoas
                                atualMesas->mesa.nOcupados = pessoas;
                                return atualMesas->mesa.idMesa;
                        }
                        else {
                                atualMesas = atualMesas->seg;
                        }
                }
                // não há mesas livres
                return MESA_VAZIA;
        }
}

/*
```

```
* Alocar pessoas às mesas
*/
void atribuiMesaGrupo(TSListaClientes* clientes, TSListaMesas* mesas) {

	TSListaClientes* atualClientes = clientes;

	int topologiaAtual = 0;

	if (atualClientes == NULL || mesas == NULL) {
		printf("\n\n##--- Lista vazia ---##\n\n");
		return;
	}
	else {//percorer a listas dos clientes
		while (atualClientes != NULL) {
			// determinar a tipologia da mesa a atribuir ao cliente
			topologiaAtual = topologiaCliente(atualClientes-
>cliente.pessoas);
			if (topologiaAtual != MESA_VAZIA) {
				//tentar atribuir mesa ao grupo de clientes
				atualClientes->cliente.mesaAtribuida =
sentarGrupo(mesas, topologiaAtual, atualClientes->cliente.pessoas);
			}
			else {
				printf("Grupo com numero de pessoas inválido!!!");
			}
			atualClientes = atualClientes->seg;
		}
	}
}
```

**B) comentário explicativo da solução (1 parágrafo).**

Foi utilizado uma lista simplesmente ligada para guardar a informação dos grupos e outra para a informação das mesas. Junto dos clientes foi adicionado um campo para a identificação da mesa atribuída. O algoritmo da distribuição determina para cada grupo qual a menor topologia de mesa que se adequa ao número de pessoas desse grupo e se existir uma mesa livre com essa tipologia é-lhe atribuída, caso contrário, fica a aguardar. Foram criados os módulos ler_escrever(.h/.c) para tratar da leitura dos ficheiros e da listagem da alocação dos grupos, e o módulo distribuição(.h/.c) para o tratamento das listas e o algoritmo da distribuição.

Exemplo da execução do programa com a impressão de ambas as listas e da distribuição dos grupos pelas mesas.