 DEPARTAMENTO DE ENGENHARIA INFORMÁTICA E DE

SISTEMAS

INSTITUTO SUPERIOR DE ENGENHARIA DE COIMBRA

2019/2020

Programação

-Trabalho Prático -

-Relatório-

Licenciatura em Engenharia informática

Vitor Fabião nº 2017009683

# Índice

# • Introdução………………………………………………….3

# Principais Estruturas de Dados……………………………3

# • Ficheiros utilizados………………………………………..5

# • Estruturas dinâmicas implementadas………………….6

# Opções e Reflexão…………………………………………7

# • Manual de funcionamento………………………………..8

# Introdução

Pretende-se que seja implementado um programa que simule, de forma simplificada, a propagação de um vírus entre uma população. Os indivíduos da população encontram-se num espaço constituído por locais, interligados entre si. A execução do programa consiste em 2 fases sequenciais:

1. **Preparação**: No início da execução, a aplicação carrega todos os dados necessários a partir de ficheiros: Configuração do espaço e dimensão e características da população inicial.
2. **Simulação**: Quando termina a fase de preparação, inicia-se um processo de simulação iterativo que representa a propagação do vírus ao longo do tempo (1 iteração de cada vez, correspondendo cada uma destas iterações a 1 dia). No final da simulação, o programa apresenta um resumo dos resultados e grava informação relevante em ficheiros

# Principais estruturas de Dados

**Pessoas:** As pessoas estão guardadas numa lista ligada em que cada nodo contem uma estrutura deste tipo. Esta estrutura contem as variáveis que considerei mais necessárias para guardar relativamente a cada pessoa. Além dos identificadores e variáveis pessoais, contém também um ponteiro que apontará para o próximo individuo da lista.

struct Pessoas {

char nome[20];

int idade;

char estado;

int doente;

int diasMax;

ppessoas next;

};

**Sala:** As salas estão guardadas num array de memória dinâmica ao qual cada “slot” contem uma estrutura deste tipo. Esta foi uma estrutura requerida pelo professor e não teve nenhuma alteração durante o código.

struct sala {

int id; // id numérico do local

int capacidade; // capacidade máxima

int liga[3]; // id das ligações (-1 nos casos não usados)

};

**Par:** Esta foi uma estrutura criada a pensar na futura simplicidade que pudesse trazer ao código. Em vez de andar sempre com a lista de pessoas e ter que a percorrer pensei nesta solução. Esta solução é bastante simples e eficaz, visto que contem os elementos todos necessários à execução do código, de uma maneira mais organizada. Esta estrutura conta com um ponteiro para a respetiva sala atribuída, com inteiros que indicam a quantidade de pessoas na sala e também as infetadas, e conta também com um array de memória dinâmica de ponteiros para estruturas do tipo pessoas.

struct Par {

plocal sala;

int total, totalInfetadas;

ppessoas \*pessoas;

};

Para ter uma melhor noção do esquema, pode observar a figura que se segue:

Uma imagem com captura de ecrã

Descrição gerada automaticamente

**Iteração:** Esta estrutura foi especialmente criada para guardar as informações necessárias para que quando se voltasse atras numa iteração conseguir substituir tudo. Por opção excluí o array dos espaços visto que permanece igual do inicio ao fim do programa. Contém o array dos pares para não se perder a informação correspondente a quem estava em qual sala. Igualmente guardo uma cópia da lista ligada de pessoas visto que está sempre em mutação a cada iteração.

struct Iteracoes{

ppar conjuntos; // Array dinâmico dos pares

ppessoas lpessoa;

pite next;

};

Para ter uma melhor noção do esquema, pode observar a figura que se segue:

Uma imagem com captura de ecrã

Descrição gerada automaticamente

# Ficheiros Utilizados

**Preparacao.c:**  Este ficheiro ao contrário do normal, é o que contém a função main. Este ficheiro é aquele que vai dar o inicio ao programa fazendo os emparelhamentos e tem um menu de orientação para o utilizador. É este que contém a criação das variáveis principais que irão ser usadas ao longo do programa.

**Espaco.c:** Este ficheiro contem a função em que carrega os espaços para o array dinâmico e os devolve, depois tem a função que verifica se as ligações estão corretas e no final o mostra espaços.

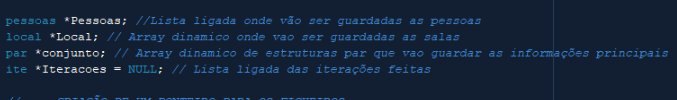
**Pessoas.c :** Este ficheiro contem a função que carrega a lista de pessoas e no fim a devolve e a também. Tem a função que permite que o utilizador possa consultar se a lista foi devidamente carregada ou não.

**Simulacao.c:** Sendo este o ficheiro que contem o maior numero de funções, é, também, um dos mais importantes visto que trata de toda a logística assim que a simulação começa. Tem um menu de orientação que conforme o numero inserido, irá proceder de modo a que a simulação avance. Este menu será explicado mais a abaixo no relatório.

**Utils.c:**  Ficheiro fornecido pelos professores em que usei todas as funções que la estavam.

**Headers.h:**  Este header contem todas as declarações das estruturas feitas. Escolhi fazê-lo desta maneira para que em casos mais à frente não houvesse dependência de informação. O que quero dizer com isto é que, se por exemplo o header do Pessoas necessitasse de informação do header do Espaco, e simultaneamente o contrario, então estaríamos num impasse. Pareceu-me ser a melhor solução.

# Estruturas dinâmicas implementadas



Na primeira variável da figura eu declarei um ponteiro para uma estrutura do tipo pessoa. Quando a função do carregaPessoas for chamada aquele ponteiro vai ser a cabeça da lista ligada. Esta lista ligada a cada iteração que passa vai ser copiada para dentro da estrutura iteração que assim desse modo conseguimos sempre aceder a um novo ponteiro.

Na segunda variável que é o ponteiro para uma estrutura do tipo Local em que quando se chama a função carrega espaço, ela retorna o array dinâmico que contem os espaços todos. Cada endereço de cada slot do array vai ser usado para ser colocado na struct Par o que vai permitir aceder sempre à sala sem ter que correr o array.

Na terceira variável da figura que é um ponteiro para uma estrutura do tipo par, quando o emparelhamento for feito o que vai acontecer é que a estrutura como tem o ponteiro para a sala, e o array de ponteiros para as pessoas, então significa que nessa estrutura que tem a sala, também vai ter todas as pessoas correspondentes a essa sala.

Na quarta variável esta a ser criado um ponteiro que vai estar a apontar posteriormente para a cabeça da lista ligada das iterações. Por opção excluí o array dos espaços visto que permanece igual do inicio ao fim do programa. Contém o array dos pares para não se perder a informação correspondente a quem estava em qual sala. Igualmente guardo uma cópia da lista ligada de pessoas visto que está sempre em mutação a cada iteração.

# Opções e reflexão

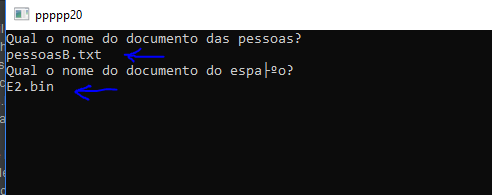
Algumas das opções que tomei foram iguais às que foram lecionadas em aulas, no entanto, noutras tive de pensar em como resolver do problema, por isso, penso que apesar de não ser a resolução perfeita, nem mais sintética, funciona.

Com isto, enumero algumas decisões que tomei:

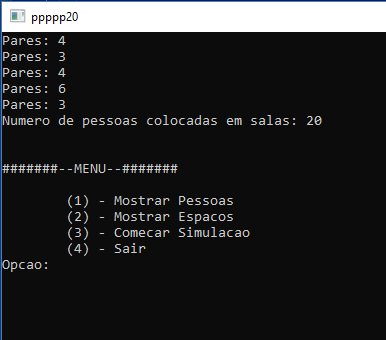
* Em vez de estar a adicionar um ponteiro para a estrutura sala dentro da estrutura pessoa, decidi criar as estrutura Par que veio facilitar, a meu ver, imensamente o trabalho. Em vez de estar sempre a correr a lista ligada das pessoas e o array das salas, sem ter que andar com isso atras, arranjei esta solução que organiza a informação de um modo muito mais simples permitindo aceder sempre da melhor forma.
* Para as iterações estive no inicio a pensar que o utilizador poderia voltar quantas vezes quisesse para trás, mas depois questionei se a dificuldade da funcionalidade não ficaria comprometida, então limitei-me ao enunciado e criei uma estrutura igualmente organizada que guardasse os ponteiros para as copias das informações. Deste modo, quando o user solicitasse para voltar atras a informação já estaria pronta a ser usada e só precisava de substituir os ponteiros.
* Na função do guarda iteração perdi algum tempo a pensar na solução mais otimizada e penso que atingi o objetivo. Nesta função questionei-me em que sentido seria a iteração mais recente (se no fim da lista ou no inicio) e então decidi que seria no fim que assim so teria que igualar a cabeça da lista ao next e estaria sempre tudo pronto e ao dispor.
* Na função de emparelhamento, inicialmente comecei por começar a correr a lista ligada e escolher uma sala aleatória para aquela pessoas até que mais à frente pensei no seguinte exemplo: Se a soma das capacidades de todas as salas for 10 e a lista ligada tiver 20 pessoas então significa que em circunstancia alguma as ultimas 10 pessoas dessa lista terão hipótese de ficar alocadas nalguma sala. Para isso percebi que teria que para além de escolher uma sala aleatória, também tinha que escolher uma pessoa aleatória.

# Manual de funcionamento

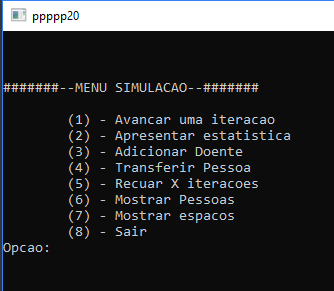
No inicio do programa é perguntado ao user qual o ficheiro das pessoas e dos espaços que pretende abrir.



De seguida, é apresentado um pequeno menu em que o user pode consultar as pessoas que foram carregadas (1) , os espaços carregados (2) , dar inicio à simulação (3) ou então sair do programa (4).



Se a pessoa der inicio à simulação irá para um outro menu no qual irá ter varias opções. Essas opções estão explicitas e o user age de acordo com o que pretende.



1. – **Avançar uma iteração** -> O user ao selecionar esta opção, irão ser aplicadas funções de tal modo que avance uma iteração com sucesso, ao faze-lo, as pessoas infetadas nessa iteração serão apresentadas no ecrã
2. – **Apresentar estatística** -> Esta opção apresenta no ecrã as pessoas distribuídas em cada uma das salas, as pessoas que foram afetadas na ultima iteração e também a percentagem de imunes, infetados e saudáveis por sala.
3. – **Adicionar Doente** -> Esta função permite ao user acrescentar um doente. Vai ser o mesmo a escolher o ID da sala, nome, idade, dias que já esta infetado.
4. – **Transferir Pessoa** -> Esta opção permite que possam ser transferidas X pessoas da Sala Z para a Sala Y. O user basta indicar estes 3 aspetos e se corretos, então o programa procede à transferência
5. – **Recurar X iterações** -> Nesta opção o o user apenas poderá indicar quantas iterações, entre 1 e 3, pretende recuar. Se válidas então o program recua 3 iterações.
6. – **Mostra Pessoas** -> Função que mostra a informação das pessoas detalhada naquele momento.
7. – **Mostra Espaços** -> Função que mostra detalhadamente a informação dos locais existentes.
8. – **Sair** -> Função que termina o programa e quando procede aos free’s.