**Uma imagem com texto, verde, captura de ecrã, Tipo de letra

Os conteúdos gerados por IA poderão estar incorretos.**TRabalho EStrutura de Dados

Fase 1

EStrutura de Dados

FAse 1

RESUMO

O presente relatório descreve o desenvolvimento de um projeto no âmbito da Unidade Curricular (UC) de Estruturas de Dados Avançadas (EDA), integrada no 2.º semestre do 1.º ano do curso. O objetivo deste projeto é aplicar e consolidar os conhecimentos adquiridos ao longo do semestre, através da implementação de estruturas de dados dinâmicas na linguagem de programação C.

Neste relatório, serão detalhadas as etapas de desenvolvimento do projeto, incluindo a definição das estruturas de dados, a implementação das funcionalidades essenciais, e a análise dos resultados obtidos. Além disso, será descrito o processo de armazenamento e carregamento de dados a partir de ficheiros, bem como a organização modular do código e a sua documentação com Doxygen.

Este documento visa, assim, não apenas apresentar a solução desenvolvida, mas também discutir as decisões tomadas ao longo do processo, os desafios encontrados e as soluções adotadas para os ultrapassar.

Índice

[RESUMO 3](#_Toc194248251)

[Índice de figuras 7](#_Toc194248252)

[**Introdução** 9](#_Toc194248253)

[Capítulo 1 - Abordagem Inicial e Mudança de Estratégia 11](#_Toc194248254)

[1.1-Abordagem Inicial 11](#_Toc194248255)

[1.2-Abordagem Final Adotada 13](#_Toc194248256)

[Capítulo 2 - Estruturas de Dados e Implementação 15](#_Toc194248257)

[2.1-Estrutura para armazenar antenas 16](#_Toc194248258)

[2.2-Estrutura para armazenar os efeitos nefastos 17](#_Toc194248259)

[2.3-Variaveis Globais 18](#_Toc194248260)

[2.4-Classe Diagram 19](#_Toc194248261)

[Capítulo 3 - Funções do programa 21](#_Toc194248262)

[3.1-Funções do programa 22](#_Toc194248263)

[3.1.1-Função criar Antena 22](#_Toc194248264)

[3.1.2-Função Carregar Antena do ficheiro 24](#_Toc194248265)

[3.1.3-Função Procura Antena 28](#_Toc194248266)

[3.1.4-Função Inserir Antena através de coordenadas 29](#_Toc194248267)

[3.1.5-Função Remover Antena através de coordenadas 34](#_Toc194248268)

[Capítulo 4 - MAIN 50](#_Toc194248269)

[Capítulo 4 – Programa 54](#_Toc194248270)

[Menu 54](#_Toc194248271)

[Carregar antenas 54](#_Toc194248272)

[Mostrar matriz de antenas 55](#_Toc194248273)

[Calcular e mostra efeitos nefastos 55](#_Toc194248274)

[Inserir nova antena 56](#_Toc194248275)

[Remover antena 57](#_Toc194248276)

[Conclusão 59](#_Toc194248277)

# Índice de figuras

[Figura 1Classe Diagram 19](#_Toc194248227)

[Figura 2 Assinatura das Funções 21](#_Toc194248228)

[Figura 3 Menu do programa 54](#_Toc194248229)

[Figura 4 Função Carregar Antenas 54](#_Toc194248230)

[Figura 5 Função mostrar matriz de antenas 55](#_Toc194248231)

[Figura 6 Função mostrar efeitos nefastos 55](#_Toc194248232)

[Figura 7 Função inserir nova antena 56](#_Toc194248233)

[Figura 8 Função remover antena 57](#_Toc194248234)

# **Introdução**

A temática abordada envolve a modelação e manipulação de antenas numa cidade, onde cada antena possui uma frequência específica e pode gerar efeitos nefastos em determinadas localizações. Para representar e processar esta informação, recorreu-se ao uso de listas ligadas, permitindo armazenar e manipular dinamicamente os dados das antenas e das áreas afetadas.

A problemática em análise consiste na disposição de antenas num espaço bidimensional e na identificação das localizações afetadas pelo efeito nefasto causado pela ressonância de frequências iguais. Para tal, é necessário definir uma estrutura de dados que permita gerir dinamicamente as antenas e as localizações com efeito nefasto, garantindo uma organização eficiente da informação.

Inicialmente, a abordagem prevista no enunciado para o cálculo das localizações com efeito nefasto revelou-se complexa de implementar na prática. Face a esta dificuldade, optei por uma estratégia alternativa para determinar as áreas afetadas, garantindo, ainda assim, a deteção eficiente do efeito nefasto causado pela ressonância das antenas.

# Capítulo 1 - Abordagem Inicial e Mudança de Estratégia

## 1.1-Abordagem Inicial

Inicialmente, considerei utilizar uma matriz dinâmica de ponteiros para armazenar as antenas, representada pela estrutura antena\*\*\* matriz. A ideia por trás desta abordagem era facilitar a visualização dos dados no terminal e permitir um acesso direto às antenas com base nas suas coordenadas na matriz.

Para implementar esta abordagem, desenvolvi a função criarMatrizAntenas(int maxLinhas, int maxColunas), que aloca dinamicamente uma matriz tridimensional de ponteiros para armazenar objetos do tipo antena. Cada célula da matriz continha um ponteiro para uma estrutura antena, armazenando a sua posição (linha e coluna) e a respetiva frequência. Além disso, foi incluído um ponteiro next, que poderia ser utilizado para futuras expansões, como a gestão de colisões ou armazenamento de múltiplas antenas na mesma posição.

1 antena\*\*\* criarMatrizAntenas(int maxLinhas, int maxColunas) {

2 antena\*\*\* matriz = (antena\*\*\*)malloc(maxLinhas \* sizeof(antena\*\*));

3 if (!matriz) {

4 return NULL;

5 }

6

7 for (int i = 0; i < maxLinhas; i++) {

8 matriz[i] = (antena\*\*)malloc(maxColunas \* sizeof(antena\*));

9 if (!matriz[i]) {

10 return NULL;

11 }

12 for (int j = 0; j < maxColunas; j++) {

13 matriz[i][j] = (antena\*)malloc(sizeof(antena));

14 if (!matriz[i][j]) {

15 return NULL;

16 }

17 matriz[i][j]->linha = i;

18 matriz[i][j]->coluna = j;

19 matriz[i][j]->frequencia = ' ';

20 matriz[i][j]->next = NULL;

21 }

22 }

23

24 return matriz;

25 }

Apesar de a matriz permitir um acesso direto às antenas pelas suas coordenadas, esta abordagem revelou-se mais complexa do que o necessário. A alocação dinâmica de uma matriz tridimensional aumentava o consumo de memória e tornava a manipulação dos dados mais difícil, especialmente ao adicionar e remover antenas. Além disso, gerir os ponteiros corretamente e garantir a libertação de memória era uma preocupação adicional.

Dada a complexidade crescente da implementação e a dificuldade em manipular os dados de forma eficiente, decidi abandonar esta abordagem e procurar uma solução mais simples e adaptável, utilizando listas ligadas.

## 1.2-Abordagem Final Adotada

Após avaliar as dificuldades da abordagem inicial baseada em matrizes dinâmicas, optei por uma solução mais simples e eficiente: o uso de uma lista ligada para armazenar as antenas. Esta estrutura permite um armazenamento dinâmico dos dados, reduzindo o consumo de memória e facilitando a inserção, remoção e manipulação das antenas sem a necessidade de gerir matrizes complexas.

A implementação baseia-se numa lista ligada onde cada nó representa uma antena, contendo a sua posição (coluna e linha), a frequência e um ponteiro para o próximo elemento da lista. A criação de uma nova antena é realizada através da função CriarAntena, que aloca dinamicamente memória para uma estrutura do tipo antena e inicializa os seus valores.

1 antena\* CriarAntena(int coluna, int linha, char frequencia) {

2 antena\* aux;

3 aux = (antena\*)malloc(sizeof(antena));

4 if (aux != NULL) {

5 aux->coluna = coluna;

6 aux->linha = linha;

7 aux->frequencia = frequencia;

8 aux->next = NULL;

9 }

10 return aux;

11 }

Esta abordagem mostrou-se mais vantajosa relação à matriz tridimensional inicialmente considerada:

1. **Otimização da alocação de memória** – A memória é alocada apenas para as antenas existentes, evitando o desperdício associado à reserva de espaço para uma matriz de tamanho fixo.
2. **Simplicidade na manipulação dos dados** – A estrutura de lista ligada permite adicionar e remover antenas de forma direta, sem necessidade de gerir índices complexos ou deslocar elementos, como seria necessário numa matriz.
3. **Maior flexibilidade** – Novas antenas podem ser adicionadas dinamicamente sem necessidade de realocar grandes blocos de memória, tornando a solução mais eficiente e adaptável a diferentes dimensões do problema.

Assim, a utilização de uma lista ligada revelou-se uma abordagem mais adequada, permitindo uma implementação mais clara, modular e eficiente.

# Capítulo 2 - Estruturas de Dados e Implementação

Nesta secção são descritas as estruturas de dados utilizadas para representar as antenas e as localizações com efeito nefasto, e a declaração de duas variáveis globais que são usadas ao longo do código. As estruturas e as variáveis globais foram definidas num ficheiro de cabeçalho (.h) para garantir uma melhor organização do código e facilitar a sua reutilização em diferentes partes do programa.

## 2.1-Estrutura para armazenar antenas

As antenas são representadas através de uma **lista ligada**, onde cada nó contém a informação essencial sobre a antena, nomeadamente:

* **Coluna e linha** – Coordenadas da antena na matriz que representa a cidade.
* **Frequência** – Caracter associado à frequência da antena.
* **Ponteiro para o próximo elemento (next)** – Utilizado para ligar cada antena à seguinte, formando uma estrutura dinâmica.

A estrutura foi definida no ficheiro de cabeçalho Structs.h da seguinte forma:

1 typedef struct antena {

2 int coluna;

3 int linha;

4 char frequencia;

5 struct antena\* next;

6 } antena;

Nesta estrutura, o campo next é um **ponteiro** que armazena o endereço da próxima antena na lista. Isto significa que, em vez de armazenar todas as antenas numa matriz fixa, cada antena sabe onde está a seguinte, criando assim uma **cadeia de elementos interligados**. Facilitando o uso e manipulação das antenas na estrutura de dados e tornando as operações de inserção, remoção e procura mais eficientes.

## 2.2-Estrutura para armazenar os efeitos nefastos

De forma semelhante às antenas, as localizações afetadas pelo efeito nefasto também são armazenadas numa lista ligada, garantindo flexibilidade e um uso eficiente da memória.

Cada nó desta lista contém:

* **nx e ny** – Coordenadas da localização afetada.
* **Ponteiro para o próximo elemento(next)** – Permite percorrer a lista e aceder a todas as localizações armazenadas.

A estrutura foi definida no mesmo ficheiro de cabeçalho Structs.h da seguinte forma:

1 typedef struct nefasto {

2 int nx;

3 int ny;

4 struct nefasto\* next;

5 } nefasto;

## 2.3-Variaveis Globais

int MAX\_LINHAS = 0;

int MAX\_COLUNAS = 0;

Função das variáveis globais

MAX\_LINHAS e MAX\_COLUNAS armazenam, respetivamente, o número máximo de linhas e colunas da matriz onde as antenas estão distribuídas.

Estas variáveis são utilizadas para garantir que as operações realizadas sobre as antenas respeitam os limites da matriz.

As variáveis globais são variáveis declaradas fora de qualquer função e que podem ser acedidas por qualquer parte do programa. Ao contrário das variáveis locais, que existem apenas dentro da função onde foram definidas, as variáveis globais mantêm o seu valor ao longo de toda a execução do programa.

## 2.4-Classe Diagram

Uma imagem com texto, captura de ecrã, cartão de visita, Tipo de letra

Os conteúdos gerados por IA poderão estar incorretos.

Figura 1Classe Diagram

O diagrama de classes representa a estrutura das principais entidades do projeto:

* **antena**: Estrutura que armazena a informação de cada antena, incluindo as suas coordenadas (**coluna** e **linha**), a **frequência** e um ponteiro (**next**) que liga ao próximo elemento da lista ligada.
* **nefasto**: Estrutura que representa as localizações afetadas pelo efeito nefasto, armazenando as coordenadas (**nx** e **ny**) e um ponteiro (**next**) para permitir a ligação entre os diferentes elementos na lista ligada.

# Capítulo 3 - Funções do programa

Este capítulo apresenta as principais funções desenvolvidas para o programa, detalhando a sua implementação e propósito. Estas funções permitem a gestão das antenas, a identificação de efeitos nefastos e a visualização dos dados.

A implementação destas funcionalidades segue uma abordagem modular, utilizando listas ligadas para armazenar as antenas e os pontos afetados pelos efeitos nefastos. Além disso, o programa inclui funções de carregamento de ficheiros, manipulação de antenas (inserção, remoção e pesquisa), cálculo dos efeitos nefastos e exibição dos resultados em formato matricial.

As assinaturas das funções usadas encontram-se no ficheiro ASSFunco.h.

Uma imagem com texto, captura de ecrã, Tipo de letra, software

Os conteúdos gerados por IA poderão estar incorretos.

Figura 2 Assinatura das Funções

## 3.1-Funções do programa

### 3.1.1-Função criar Antena

8 /\*\*

9 \* .

10 \* CriarAntena

11 \* \param coluna

12 \* \param linha

13 \* \param frequencia

14 \* \return

15 \*/

16 antena\* CriarAntena(int coluna, int linha, char frequencia) {

17 antena\* aux;

18 aux = (antena\*)malloc(sizeof(antena));

19 if (aux != NULL) {

20 aux->coluna = coluna;

21 aux->linha = linha;

22 aux->frequencia = frequencia;

23 aux->next = NULL;

24 }

25 return aux;

26

27

28 }

Esta função tem como objetivo criar uma estrutura de dados do tipo antena, que é alocada dinamicamente na memória. A função recebe três parâmetros: coluna, linha, e frequência, que vão ser usados para inicializar a nova antena. A função retorna um ponteiro para a estrutura antena criada.

**Partes a destacar:**

18 aux = (antena\*)malloc(sizeof(antena));

* malloc: A função malloc é usada para alocar um bloco de memória do tamanho necessário para armazenar a estrutura antena. No caso, a memória alocada é do tamanho de uma estrutura antena, que é obtido através de sizeof(antena).

19 if (aux != NULL) {

* Verificação de falha na alocação: Após chamar malloc, é importante verificar se a memória foi alocada corretamente. Se malloc falhar, ela retorna NULL, por isso a condição aux != NULL verifica se a alocação foi bem-sucedida.
* Tratamento de erro: Caso malloc falhe (isto é, se aux for NULL), a função não vai inicializar a estrutura e simplesmente retornará NULL. Isso impede que o programa tente acessar uma área de memória não alocada, o que poderia levar a erros de execução.

20 aux->coluna = coluna;

21 aux->linha = linha;

22 aux->frequencia = frequencia;

23 aux->next = NULL;

* Atribuições dos valores: Se a alocação for bem-sucedida, os valores dos parâmetros coluna, linha, e frequência são atribuídos às variáveis correspondentes na estrutura antena. Isto é feito através do ponteiro aux, usando o operador ->, que é utilizado para acessar membros de uma estrutura através de um ponteiro.
* Inicialização de next: A estrutura antena tem um campo next, que indica o próximo elemento em uma lista encadeada. O valor de next é inicializado para NULL, indicando que, neste momento, a antena não está ligada a nenhum outro elemento.

25 return aux;

* Retorno: A função retorna o ponteiro aux, que aponta para a nova estrutura antena criada e inicializada.

### 3.1.2-Função Carregar Antena do ficheiro

30 /\*\*

31 \* .

32 \* CarregarAntenasDoFicheiro

33 \* \param nome\_ficheiro

34 \* \return

35 \*/

36 antena\* CarregarAntenasDoFicheiro(const char\* nome\_ficheiro) {

37 FILE\* fp = fopen(nome\_ficheiro, "r");

38 if (fp == NULL) {

39 printf("Erro ao abrir\n");

40 return NULL;

41 }

42

43 antena\* lista = NULL;

44 antena\* ultima = NULL;

45 char linha[256];

46 int l = 0;

47 MAX\_COLUNAS = 0;

48

49

50 while (fgets(linha, sizeof(linha), fp)) {

51 int comp = strlen(linha);

52

53

54 if (linha[comp - 1] == '\n') {

55 linha[comp - 1] = '\0';

56 comp--;

57 }

58

59 if (comp > MAX\_COLUNAS) {

60 MAX\_COLUNAS = comp;

61 }

62

63 for (int c = 0; c < comp; c++) {

64 char fre = linha[c];

65

66

67 if (fre != '.') {

68 antena\* ant = CriarAntena(c, l, fre);

69

70

71 if (lista == NULL) {

72 lista = ant;

73 }

74 else {

75 ultima->next = ant;

76 }

77 ultima = ant;

78 }

79 }

80 l++;

81 }

82 MAX\_LINHAS = l;

83 fclose(fp);

84 return lista;

85 }

Esta função tem como objetivo carregar as antenas de um ficheiro de texto e criar uma lista encadeada de estruturas antena. Cada linha do ficheiro representa uma linha de antenas, e cada caracter dentro da linha é uma antena. A função devolve um ponteiro para o início da lista encadeada de antenas.

**Abertura do Ficheiro:**

37 FILE\* fp = fopen(nome\_ficheiro, "r");

38 if (fp == NULL) {

39 printf("Erro ao abrir\n");

40 return NULL;

41 }

* **fopen:** A função fopen é usada para abrir o ficheiro no modo leitura ("r"). Se o ficheiro não puder ser aberto (por exemplo, se o ficheiro não existir), fopen retorna NULL.
* **Verificação de erro:** Se fp for NULL, a função imprime uma mensagem de erro e retorna NULL, indicando que não foi possível abrir o ficheiro.

**Inicialização das Variáveis:**

43 antena\* lista = NULL;

44 antena\* ultima = NULL;

45 char linha[256];

46 int l = 0;

47 MAX\_COLUNAS = 0;

* **lista:** Ponteiro para o início da lista encadeada de antenas, inicializado como NULL.
* **ultima:** Ponteiro usado para acompanhar o último elemento da lista encadeada, também inicializado como NULL.
* **linha:** Um array de caracteres usado para armazenar cada linha lida do ficheiro.
* **l:** Variável que representa o número de linhas lidas (usada para posicionar as antenas na "linha" correta).
* **MAX\_COLUNAS:** Variável que armazena o número máximo de colunas lidas até o momento.

**Leitura Linha a Linha:**

50 while (fgets(linha, sizeof(linha), fp)) {

51 int comp = strlen(linha);

* **fgets:** A função fgets lê uma linha do ficheiro e armazena no array linha. Ela continua a ler até encontrar uma nova linha ou atingir o limite de tamanho do array (sizeof(linha)).
* **strlen:** A função strlen calcula o comprimento da linha lida (em número de caracteres).

**Remoção do Caractere de Nova Linha:**

54 if (linha[comp - 1] == '\n') {

55 linha[comp - 1] = '\0';

56 comp--;

57 }

* Este bloco verifica se o último caractere da linha lida é um caractere de nova linha ('\n'). Se for, ele substitui o '\n' por '\0’, removendo o caractere de nova linha do final da string.

**Atualização do Número Máximo de Colunas:**

59 if (comp > MAX\_COLUNAS) {

60 MAX\_COLUNAS = comp;

61 }

* Nesta parte o valor de MAX\_COLUNAS é atualizado se o comprimento da linha lida (comp) for maior do que o valor atual de MAX\_COLUNAS.

**Leitura Caractere a Caractere:**

63 for (int c = 0; c < comp; c++) {

64 char fre = linha[c];

* Este laço percorre cada caractere na linha lida e atribui o caractere à variável fre.

**Criação das Antenas:**

67 if (fre != '.') {

68 antena\* ant = CriarAntena(c, l, fre);

* Se o caractere não for um ponto (.), significa que ele representa uma antena. A função CriarAntena é chamada para criar uma nova antena, passando a posição c (coluna), l (linha) e a frequência (o caractere lido).

**Construção da Lista Encadeada:**

71 if (lista == NULL) {

72 lista = ant;

73 }

74 else {

75 ultima->next = ant;

76 }

77 ultima = ant;

* Se a lista ainda estiver vazia (lista == NULL), a primeira antena criada será atribuída ao ponteiro lista, tornando-se o início da lista.
* Se a lista já contiver elementos, a nova antena será ligada ao final da lista, atualizando o campo next da antena anterior (ultima->next = ant).
* O ponteiro ultima é atualizado para apontar para a nova antena, tornando-a a última da lista.

**Incremento do Número de Linhas:**

80 l++;

* A variável l é incrementada após processar uma linha, indicando que uma nova linha foi lida do ficheiro.

**10. Fecho do Ficheiro:**

83 fclose(fp);

* Após ler todas as linhas do ficheiro, o ficheiro é fechado usando fclose.

**11. Retorno da Lista de Antenas:**

84 return lista;

* A função retorna o ponteiro lista, que aponta para o início da lista encadeada de antenas.

### 3.1.3-Função Procura Antena

86 /\*\*

87 \* .

88 \* procurarantena

89 \* \param h

90 \* \param l

91 \* \param c

92 \* \return

93 \*/

94 antena\* procurarantena(antena\* h, int l, int c) {

95 antena\* aux = h;

96 while (aux != NULL) {

97 if (aux->linha == l && aux->coluna == c) {

98 return aux;

99 }

100 aux = aux->next;

101 }

102 return NULL;

103}

A função procurarantena tem como objetivo procurar uma antena específica na lista encadeada de antenas. Ela percorre a lista e retorna um ponteiro para a antena que estiver na posição (l, c), onde l representa a linha e c a coluna. Se a antena não for encontrada, a função retorna NULL.

**Parâmetros:**

* h: Ponteiro para o início da lista encadeada de antenas.
* l: Número da linha onde se pretende procurar a antena.
* c: Número da coluna onde se pretende procurar a antena.

**Inicialização da busca:**

antena\* aux = h;

* Um ponteiro aux é criado e inicializado com o ponteiro h, que aponta para o início da lista encadeada.

**Percorrer a lista até encontrar a linha e coluna desejada:**

97 if (aux->linha == l && aux->coluna == c) {

98 return aux;

99 }

100 aux = aux->next;

* O if percorre a lista encadeada até encontrar a linha e coluna desejada, se não encontra aux passa para o proximo até aux ser NULL (fim da lista).

**Retornar NULL caso a antena não seja encontrada:**

return NULL;

* Se não encontrarmos uma antena na posição (l, c), a função retorna NULL.

### 3.1.4-Função Inserir Antena através de coordenadas

/\*\*

113 \* .

114 \* insereOrdenado

115 \* \param inicio

116 \* \param novo

117 \* \return

118 \*/

119 antena\* insereOrdenado(antena\* inicio, antena\* novo) {

120

121 if (novo == NULL) return inicio;

122

123

124 if (procurarantena(inicio, novo->linha, novo->coluna) != NULL) {

125 free(novo);

126 return inicio;

127 }

128

129 if (inicio == NULL) {

130 inicio = novo;

131 return inicio;

132 }

133

134

135 if ((novo->linha < inicio->linha) || (novo->linha == inicio->linha && novo->coluna < inicio->coluna)) {

136 novo->next = inicio;

137 inicio = novo;

138 return inicio;

139 }

140

141 antena\* atual = inicio;

142 antena\* anterior = atual;

143

144 while ((atual->linha < novo->linha ||(atual->linha == novo->linha && atual->coluna < novo->coluna))&& atual->next != NULL) {

145

146 anterior = atual;

147 atual = atual->next;

148

149 }

150 if ((atual->linha == novo->linha && atual->coluna > novo->coluna)) {

151 novo->next = atual;

152 anterior->next = novo;

153

154 }

155 /\*final\*/

156 else if (atual->linha < novo->linha && atual->coluna < novo->coluna) {

157 atual->next = novo;

158

159 }

160 else {

161 novo->next = atual;

162 anterior->next = novo;

163 }

164 return inicio;

165 }

A função insereOrdenado tem como objetivo inserir uma nova antena (novo) numa lista encadeada de antenas (inicio), de forma ordenada. A ordem de inserção é determinada pela linha e coluna da nova antena a inserir. A função verifica se a antena já existe na lista e, em caso de já existir, não a insere. Se a lista estiver vazia, a nova antena é inserida no início.

**Parâmetros:**

* inicio: Ponteiro para o início da lista encadeada de antenas.
* novo: Ponteiro para a nova antena a ser inserida na lista.

**Verificação se a antena já existe na lista:**

if (procurarantena(inicio, novo->linha, novo->coluna) != NULL) {

free(novo);

return inicio;

}

* A função procurarantena é chamada para verificar se já existe uma antena na lista na posição (linha, coluna) da nova antena.
* Se a antena já existir, a função libera a memória alocada para novo e retorna a lista sem adicionar a nova antena.

I**nserção quando a lista está vazia:**

if (inicio == NULL) {

inicio = novo;

return inicio;

}

* Se a lista estiver vazia (ou seja, inicio for NULL), a nova antena se torna o primeiro elemento da lista. O ponteiro inicio é atualizado para apontar para novo.

**Inserção no início da lista:**

if ((novo->linha < inicio->linha) || (novo->linha == inicio->linha && novo->coluna < inicio->coluna)) {

novo->next = inicio;

inicio = novo;

return inicio;

}

* A nova antena é inserida no início da lista se ela for "menor" do que a primeira antena da lista. A comparação é feita primeiro pela linha (novo->linha < inicio->linha), e se as linhas forem iguais, pela coluna (novo->coluna < inicio->coluna).
* Se a condição for verdadeira, o ponteiro next da nova antena (novo->next) aponta para o primeiro elemento da lista (inicio), e inicio é atualizado para apontar para novo.

**Percorrer a lista para encontrar a posição correta:**

antena\* atual = inicio;

antena\* anterior = atual;

while ((atual->linha < novo->linha || (atual->linha == novo->linha && atual->coluna < novo->coluna)) && atual->next != NULL) {

anterior = atual;

atual = atual->next;

}

* Se a nova antena não for inserida no início, a função começa a percorrer a lista. São usados para percorrer a lista dois ponteiros, atual e anterior, e localizar a posição correta onde a nova antena deve ser inserida.
* O laço while percorre a lista até encontrar uma antena cujo valor da linha e coluna seja maior do que o da nova antena ou até o final da lista (atual->next != NULL).

**Inserção entre dois elementos existentes na lista:**

if ((atual->linha == novo->linha && atual->coluna > novo->coluna)) {

novo->next = atual;

anterior->next = novo;

}

* Se encontramos um elemento com a mesma linha e uma coluna maior do que a da nova antena, a nova antena é inserida entre anterior e atual. O campo next de novo aponta para atual, e o campo next de anterior aponta para novo.

**Inserção no final da lista:**

else if (atual->linha < novo->linha && atual->coluna < novo->coluna) {

atual->next = novo;

}

* Se a nova antena é maior do que todos os elementos da lista (tanto em linha quanto em coluna), ela é inserida no final da lista. O campo next de atual (o último elemento da lista) passa a apontar para a nova antena.

**Caso de inserção após um elemento, mas não no final:**

else {

novo->next = atual;

anterior->next = novo;

}

* Este bloco de código cobre o caso em que a nova antena deve ser inserida entre dois elementos da lista, mas não é a menor nem a maior. O ponteiro next de novo aponta para atual, e o ponteiro next de anterior aponta para novo.

### 3.1.5-Função Remover Antena através de coordenadas

169 /\*\*

170 \* .

171 \* removerantena

172 \* \param h

173 \* \param linha

174 \* \param coluna

175 \* \return

176 \*/

177 antena\* removerantena(antena\* h, int linha, int coluna) {

178

179 if (h == NULL) {

180 return NULL;

181 }

182 if (procurarantena(h, linha, coluna) == NULL) return NULL;

183

184

185

186 if (h->linha == linha && h->coluna == coluna) {

187 antena\* aux = h;

188 h = h->next;

189 free(aux);

190 return h;

191 }

192

193 antena\* aux = h;

194 antena\* auxAnt = aux;

195

196 while (aux && (aux->linha != linha || aux->coluna != coluna)) {

197 auxAnt = aux;

198 aux = aux->next;

199 }

200

201 if (aux ) {

202 auxAnt->next = aux->next;

203 free(aux);

204 }

205

206 return h;

207 }

A função removerantena tem como objetivo remover uma antena da lista encadeada de antenas com base nas coordenadas (linha, coluna). Caso a antena seja encontrada, ela é removida da lista e a memória é liberada. Se a antena não existir na lista, a função retorna a lista original sem modificações.

**Parâmetros:**

* h: Ponteiro para o início da lista encadeada de antenas.
* linha: Coordenada da linha da antena que se pretende remover.
* coluna: Coordenada da coluna da antena que se pretende remover.

**Verificar se a lista está vazia:**

if (h == NULL) {

return NULL;

}

* Se h for NULL, significa que a lista está vazia. Nesse caso, não há nada para remover e a função retorna NULL.

**Verificar se a antena a ser removida existe na lista:**

if (procurarantena(h, linha, coluna) == NULL) return NULL;

* A função procurarantena é chamada para verificar se há uma antena na posição (linha, coluna).
* Se a antena não existir, a função retorna a lista original (h), pois não há nada para remover.

**Remover a antena que está no início da lista:**

if (h->linha == linha && h->coluna == coluna) {

antena\* aux = h;

h = h->next;

free(aux);

return h;

}

* Se a antena a ser removida estiver no início da lista (h->linha == linha && h->coluna == coluna), um ponteiro auxiliar aux armazena o nó a ser removido.
* O ponteiro h é atualizado para apontar para o próximo nó (h = h->next).
* A memória ocupada pela antena removida é liberada (free(aux)).
* A função retorna o novo início da lista.

**Percorrer a lista para encontrar a antena a ser removida:**

antena\* aux = h;

antena\* auxAnt = aux;

while (aux && (aux->linha != linha || aux->coluna != coluna)) {

auxAnt = aux;

aux = aux->next;

}

* Dois ponteiros são usados para percorrer a lista:
  + aux: percorre a lista procurando a antena a ser removida.
  + auxAnt: armazena o nó anterior ao nó que será removido.
* O laço while continua até aux encontrar a antena com as coordenadas (linha, coluna) ou até aux ser NULL (fim da lista).

**Remover a antena encontrada:**

if (aux) {

auxAnt->next = aux->next;

free(aux);

}

* Se aux não for NULL, significa que encontramos a antena a ser removida.
* O ponteiro next do nó anterior (auxAnt) passa a apontar para o próximo nó após aux, removendo aux da lista.
* A memória da antena removida é liberada (free(aux)).

3.1.6-Função para Mostrar a Matriz das antenas

/\*\*

\* mostrarMatrizAntenas

\* \param lista\_antenas Lista de antenas carregadas

\*/

224 void mostrarMatrizAntenas(antena\* lista) {

225 printf("\nMatriz de Antenas (%dx%d):\n", MAX\_LINHAS, MAX\_COLUNAS);

226

227 for (int l = 0; l < MAX\_LINHAS; l++) {

228 for (int c = 0; c < MAX\_COLUNAS; c++) {

229 antena\* ant = procurarantena(lista, l, c);

230 if (ant != NULL) {

231 printf("%c ", ant->frequencia); // Imprime a antena com um espaço

232 }

233 else {

234 printf(". "); // Imprime ponto com espaço para alinhamento correto

235 }

236 }

237 printf("\n");

238 }

239 }

A função mostrarMatrizAntenas() imprime no ecrã uma representação visual em formato de matriz (linhas × colunas) das antenas armazenadas numa lista ligada. Cada posição da matriz mostra:

* **Uma letra** (frequência da antena, como A, B, etc.) se existir uma antena nessa posição.
* **Um ponto (.)** se a posição estiver vazia.

**Parâmetros:**

Lista-Ponteiro para o início da lista ligada de antenas.

**Percorrer a Matriz**

for (int l = 0; l < MAX\_LINHAS; l++) {

for (int c = 0; c < MAX\_COLUNAS; c++) {

* Dois loops aninhados percorrem todas as posições da matriz:
  + **l** (linha) varia de 0 a MAX\_LINHAS-1.
  + **c** (coluna) varia de 0 a MAX\_COLUNAS-1.

**Verificação de Antenas**

antena\* ant = procurarantena(lista, l, c);

Para cada posição (l, c), a função procurarantena() verifica se existe uma antena na lista ligada.

Retorna um ponteiro para a antena (ant) ou NULL se não houver antena nessa posição.

**Impressão da Matriz**

if (ant != NULL) {

printf("%c ", ant->frequencia); // Mostra a letra da antena

}

else {

printf(". "); // Mostra um ponto se estiver vazio

}

Se existir uma antena (ant != NULL), imprime a sua frequência (ex: A ).

Caso contrário, imprime para indicar posição vazia.

O espaço após cada carácter garante alinhamento correto entre colunas.

3.1.7-Função para libertar a memoria usada para Antenas

241 void libertarListaAntenas(antena\* lista) {

242 antena\* atual = lista;

243

244 while (atual != NULL) {

245 antena\* proximo = atual->next;

246 free(atual);

247 atual = proximo;

248 }

249}

Esta função liberta toda a memória alocada da lista ligada de antenas, evitando deixar lixo na memoria. Percorre a lista **nó a nó**, libertando cada antena.

**Parâmetros:**

lista-Ponteiro para o primeiro elemento da lista de antenas.

**Ciclo que passa por toda a lista e liberta a memoria**

while (atual != NULL) {

245 antena\* proximo = atual->next;

246 free(atual);

247 atual = proximo;

248 }

O ciclo continua enquanto atual não for NULL (ou seja, até chegar ao fim da lista).

Guarda temporariamente o endereço do próximo nó (atual->next) antes de libertar o nó atual.

Liberta a memória alocada para o nó atual (atual).

3.1.8-Função para criar o Efeito Nefasto

/\*\*

\* Cria um novo efeito nefasto

\* \param x Coordenada x (linha) do efeito nefasto

\* \param y Coordenada y (coluna) do efeito nefasto

\* \return Ponteiro para o novo efeito nefasto criado

\*/

251 nefasto\* criarEfeitoNefasto(int x, int y) {

252 nefasto\* novo = (nefasto\*)malloc(sizeof(nefasto));

253 if (novo != NULL) {

254 novo->nx = x;

255 novo->ny = y;

256 novo->next = NULL;

257 }

258 return novo;

259 }

Esta função aloca memória e cria uma nova estrutura do tipo nefasto (ponto nefasto) com as coordenadas especificadas. (Possui a mesma logica que a função CriarAntenas)

**Parâmetros:**

x-Coordenada da linha onde o efeito ocorre

y-Coordenada da coluna onde o efeito ocorre

3.1.9-Função para Calcular o Efeito Nefasto

/\*\*

\* Calcula todos os efeitos nefastos entre antenas de mesma frequência

\* \param lista Lista de antenas a analisar

\* \return Lista de pontos nefastos encontrados

\*/

260 nefasto\* calcularEfeitoNefastoFinal(antena\* lista) {

261 nefasto\* nefastos = NULL;

262 antena\* atual = lista;

263

264 while (atual != NULL) {

265 antena\* comparar = atual->next;

266

267 while (comparar != NULL) {

268 if (atual->frequencia == comparar->frequencia) {

269

270 int diff\_linha = comparar->linha - atual->linha;

271 int diff\_coluna = comparar->coluna - atual->coluna;

272

273 // Calcula ponto na direção de comparar

274 int nefasto\_linha = comparar->linha + diff\_linha;

275 int nefasto\_coluna = comparar->coluna + diff\_coluna;

276

277 // Verifica se está dentro dos limites

278 if (nefasto\_linha >= 0 && nefasto\_coluna >= 0 &&

279 nefasto\_linha < MAX\_LINHAS && nefasto\_coluna < MAX\_COLUNAS) {

280 nefasto\* novo = criarEfeitoNefasto(nefasto\_linha, nefasto\_coluna);

281 novo->next = nefastos;

282 nefastos = novo;

283 }

284

285 // Calcula ponto na direção oposta (de atual)

286 nefasto\_linha = atual->linha - diff\_linha;

287 nefasto\_coluna = atual->coluna - diff\_coluna;

288

289 // Verifica novamente os limites

290 if (nefasto\_linha >= 0 && nefasto\_coluna >= 0 &&

291 nefasto\_linha < MAX\_LINHAS && nefasto\_coluna < MAX\_COLUNAS) {

292 nefasto\* novo2 = criarEfeitoNefasto(nefasto\_linha, nefasto\_coluna);

293 novo2->next = nefastos;

294 nefastos = novo2;

295 }

296 }

297 comparar = comparar->next;

298 }

299 atual = atual->next;

300 }

301 return nefastos;

302 }

A função calcularEfeitoNefastoFinal analisa uma lista de antenas e calcula os efeitos nefastos entre antenas que operam na mesma frequência. Ela verifica os pontos ao redor das antenas e adiciona pontos nefastos à lista de resultados quando se encaixam dentro dos limites estabelecidos. A função retorna uma lista de pontos nefastos encontrados.

O cálculo do efeito nefasto é realizado tendo em conta as coordenadas da primeira antena encontrada e a próxima, fazendo uma subtração da (x2-x1; y2-y1), e em seguida as coordenadas da primeira antena são subtraídas a esses valores e caso existam tais coordenadas o efeito nefasto é colocado, já a segunda é realizada uma soma e o efeito nefasto é colocado nas novas coordenadas.

**Parâmetros:**

lista (antena\*) - Ponteiro para a lista de antenas a ser analisada.

**Percurso do laço:**

while (atual != NULL) {

O primeiro laço (while) percorre todas as antenas da lista, com o ponteiro atual apontando para a antena corrente.

Dentro deste laço, o ponteiro comparar percorre as antenas subsequentes, começando da antena seguinte à atual.

antena\* comparar = atual->next;

while (comparar != NULL) {

**Comparação entre Antenas:**

if (atual->frequencia == comparar->frequencia) {

A condição if (atual->frequencia == comparar->frequencia) verifica se as antenas operam na mesma frequência.

**Cálculo de Efeitos Nefastos:**

Primeiro ponto nefasto (direção de comparar):

A diferença entre as linhas e as colunas das antenas é calculada.

int diff\_linha = comparar->linha - atual->linha;

int diff\_coluna = comparar->coluna - atual->coluna;

O primeiro ponto nefasto é calculado utilizando a diferença de posições.

int nefasto\_linha = comparar->linha + diff\_linha;

int nefasto\_coluna = comparar->coluna + diff\_coluna;

**Segundo ponto nefasto (direção de atual):**

O ponto nefasto na direção oposta à antena atual é calculado.

nefasto\_linha = atual->linha - diff\_linha;

nefasto\_coluna = atual->coluna - diff\_coluna;

**Verificação dos Limites:**

if (nefasto\_linha >= 0 && nefasto\_coluna >= 0 &&

nefasto\_linha < MAX\_LINHAS && nefasto\_coluna < MAX\_COLUNAS) {

Antes de adicionar um ponto à lista de pontos nefastos, a função verifica se o ponto está dentro dos limites da matriz, utilizando as constantes MAX\_LINHAS e MAX\_COLUNAS.

**Criação e Adição dos Pontos Nefastos:**

nefasto\* novo = criarEfeitoNefasto(nefasto\_linha, nefasto\_coluna);

novo->next = nefastos;

nefastos = novo;

Para o primeiro ponto, caso esteja dentro dos limites, é criado um novo ponto nefasto e é adicionado à lista nefastos.

O mesmo processo é repetido para o segundo ponto nefasto calculado.

if (nefasto\_linha >= 0 && nefasto\_coluna >= 0 &&

nefasto\_linha < MAX\_LINHAS && nefasto\_coluna < MAX\_COLUNAS) {

nefasto\* novo2 = criarEfeitoNefasto(nefasto\_linha, nefasto\_coluna);

novo2->next = nefastos;

nefastos = novo2;

}

**Movimento para a Próxima Antena:**

atual = atual->next;

Após comparar uma antena com todas as seguintes, o ponteiro atual é movido para a próxima antena da lista.

3.1.10-Função para Mostrar Matriz do Efeito Nefasto

/\*\*

\* Mostra matriz com antenas e efeitos nefastos

\* \param lista\_antenas Lista de antenas

\* \param lista\_nefastos Lista de pontos nefastos

\*/

350 void mostrarMatrizNefastos(antena\* lista\_antenas, nefasto\* lista\_nefastos) {

351 printf("\nMatriz com Efeitos Nefastos (%dx%d):\n", MAX\_LINHAS, MAX\_COLUNAS);

352

353 for (int l = 0; l < MAX\_LINHAS; l++) {

354 for (int c = 0; c < MAX\_COLUNAS; c++) {

355 antena\* ant = procurarantena(lista\_antenas, l, c);

356

357 if (ant != NULL) {

358 printf("%c ", ant->frequencia); // Mostra antena

359 }

360 else {

361 nefasto\* n = lista\_nefastos;

362 int ehnefasto = 0;

363

364 while (n != NULL && ehnefasto == 0) {

365 if (n->nx == l && n->ny == c) {

366 ehnefasto = 1; // Marca como ponto nefasto

367 }

368 n = n->next;

369 }

370

371 if (ehnefasto) {

372 printf("# "); // Mostra efeito nefasto

373 }

374 else {

375 printf(". "); // Mostra espaço vazio

376 }

377 }

378 }

379 printf("\n"); // Nova linha ao final de cada linha

380 }

381 }

A função mostrarMatrizNefastos() imprime no ecrã uma representação visual em formato de matriz (linhas × colunas) das antenas e dos pontos nefastos. Cada posição da matriz mostra:

* Uma letra (frequência da antena, como A, B, etc.) se existir uma antena nessa posição.
* Um símbolo (#) se a posição for um ponto nefasto.
* Um ponto (.) se a posição estiver vazia e não for um ponto nefasto.

**Parâmetros:**

lista\_antenas - Ponteiro para o início da lista ligada de antenas.

lista\_nefastos - Ponteiro para o início da lista ligada de pontos nefastos.

**Percorrer a Matriz:**

for (int l = 0; l < MAX\_LINHAS; l++) {

for (int c = 0; c < MAX\_COLUNAS; c++) {

// ...

}

}

Dois loops aninhados percorrem todas as posições da matriz:

l (linha) varia de 0 a MAX\_LINHAS-1.

c (coluna) varia de 0 a MAX\_COLUNAS-1.

**Verificação de Antenas:**

antena\* ant = procurarantena(lista\_antenas, l, c);

Para cada posição (l, c), a função procurarantena() verifica se existe uma antena na lista ligada.

Retorna um ponteiro para a antena (ant) ou NULL se não houver antena nessa posição.

**Impressão da Matriz:**

if (ant != NULL) {

printf("%c ", ant->frequencia);

} else {

printf(". ");.

}

Se existir uma antena (ant != NULL), imprime a sua frequência (ex: A).

**Verificação de Pontos Nefastos:**

nefasto\* n = lista\_nefastos;

int ehnefasto = 0;

while (n != NULL && ehnefasto == 0) {

if (n->nx == l && n->ny == c) {

ehnefasto = 1;

}

n = n->next;

}

if (ehnefasto) {

printf("# ");

} else {

printf(". ");

}

Se não existir uma antena, percorre a lista de pontos nefastos (lista\_nefastos).

Se encontrar um ponto nefasto na posição (l, c), imprime #.

Caso contrário, imprime ‘.’

O espaço após cada caractere garante o alinhamento correto entre colunas.

3.1.11-Função para libertar a memoria usada para Efeito Nefasto

/\*\*

\* Liberta a memória de uma lista de efeitos nefastos

\* \param lnefasto Ponteiro para a lista de efeitos nefastos

\*/

400 void libertarListaNefasto(nefasto\* lnefasto) {

401 nefasto\* atual = lnefasto; // Ponteiro para percorrer a lista

402

403 while (atual != NULL) { // Enquanto houver elementos na lista

404 nefasto\* proximo = atual->next; // Guarda referência ao próximo

405 free(atual); // Liberta o elemento atual

406 atual = proximo; // Move para o próximo elemento

407 }

408 }

Esta função liberta toda a memória alocada da lista ligada de nefasto, evitando deixar lixo na memoria. Percorre a lista **nó a nó**, libertando cada antena. Possui o mesmo raciocínio da função libertar memoria antenas.

**Parâmetros:**

lnefasto-Ponteiro para o primeiro elemento da lista de antenas.

**Loop de Libertação (Linhas 403-407)**

while (atual != NULL) {

nefasto\* proximo = atual->next;

free(atual);

atual = proximo;

}

Guarda o próximo elemento antes de liberar o atual em seguida liberta a memória do elemento atual e avança para o próximo elemento

# Capítulo 4 - MAIN

int main() {

int opcao;

antena\* lista\_antenas = NULL;

nefasto\* lista\_nefastos = NULL;

char nome\_ficheiro[100];

do {

printf("\nMenu:\n");

printf("1. Carregar antenas do ficheiro\n");

printf("2. Mostrar matriz de antenas\n");

printf("3. Calcular e mostrar efeitos nefastos\n");

printf("4. Inserir nova antena\n");

printf("5. Remover antena\n");

printf("0. Sair\n");

printf("Escolha uma opcao: ");

scanf("%d", &opcao);

switch (opcao) {

case 1:

printf("Digite o nome do ficheiro: ");

scanf("%s", nome\_ficheiro);

lista\_antenas = CarregarAntenasDoFicheiro(nome\_ficheiro);

if (lista\_antenas != NULL) {

printf("Antenas carregadas com sucesso!\n");

}

break;

case 2:

if (lista\_antenas == NULL) {

printf("Nenhuma antena carregada ainda.\n");

}

else {

mostrarMatrizAntenas(lista\_antenas);

}

break;

case 3:

if (lista\_antenas == NULL) {

printf("Nenhuma antena carregada ainda.\n");

}

else {

libertarListaNefasto(lista\_nefastos); // Libera lista anterior

lista\_nefastos = calcularEfeitoNefastoFinal(lista\_antenas);

mostrarMatrizNefastos(lista\_antenas, lista\_nefastos);

}

break;

case 4: {

int linha, coluna;

char frequencia;

printf("Digite a linha: ");

scanf("%d", &linha);

printf("Digite a coluna: ");

scanf("%d", &coluna);

printf("Digite a frequencia: ");

scanf(" %c", &frequencia);

antena\* nova = CriarAntena(coluna, linha, frequencia);

lista\_antenas = insereOrdenado(lista\_antenas, nova);

printf("Antena inserida com sucesso!\n");

break;

}

case 5: {

int linha, coluna;

printf("Digite a linha: ");

scanf("%d", &linha);

printf("Digite a coluna: ");

scanf("%d", &coluna);

lista\_antenas = removerantena(lista\_antenas, linha, coluna);

printf("Antena removida (se existia).\n");

break;

}

case 0:

printf("Saindo...\n");

break;

default:

printf("Opcao invalida!\n");

}

} while (opcao != 0);

libertarListaAntenas(lista\_antenas);

libertarListaNefasto(lista\_nefastos);

return 0;

}

A função main() implementa um menu interativo que permite ao utilizador realizar várias operações relacionadas com antenas e efeitos nefastos. O menu oferece as seguintes opções:

1. **Carregar antenas de um ficheiro**
2. **Mostrar a matriz de antenas**
3. **Calcular e mostrar efeitos nefastos**
4. **Inserir uma nova antena**
5. **Remover uma antena**
6. **Sair do programa**

**Declaração de Variáveis:**

* + opcao: Variável usada para armazenar a opção escolhida pelo utilizador.
  + lista\_antenas: Ponteiro para a lista de antenas.
  + lista\_nefastos: Ponteiro para a lista de pontos nefastos.
  + nome\_ficheiro: String para armazenar o nome do ficheiro a ser carregado.

# Capítulo 4 – Programa

## Menu

Uma imagem com texto, captura de ecrã, Tipo de letra

Os conteúdos gerados por IA poderão estar incorretos.

Figura 3 Menu do programa

## Carregar antenas

Uma imagem com texto, captura de ecrã, Tipo de letra

Os conteúdos gerados por IA poderão estar incorretos.

Figura 4 Função Carregar Antenas

## Mostrar matriz de antenas

Uma imagem com texto, captura de ecrã, Tipo de letra, design

Os conteúdos gerados por IA poderão estar incorretos.

Figura 5 Função mostrar matriz de antenas

## Calcular e mostra efeitos nefastos

Uma imagem com texto, captura de ecrã, Tipo de letra, design

Os conteúdos gerados por IA poderão estar incorretos.

Figura 6 Função mostrar efeitos nefastos

## Inserir nova antena

Uma imagem com texto, captura de ecrã, Tipo de letra, design

Os conteúdos gerados por IA poderão estar incorretos.

Figura 7 Função inserir nova antena

## Remover antena

Uma imagem com texto, captura de ecrã, Tipo de letra, design

Os conteúdos gerados por IA poderão estar incorretos.

Figura 8 Função remover antena

# Conclusão

A implementação da Fase 1 deste projeto proporcionou uma sólida consolidação dos conhecimentos teóricos sobre estruturas de dados dinâmicas, com ênfase nas listas ligadas, e permitiu aplicar esses conceitos de forma prática na linguagem C. Ao resolver o problema proposto, foi possível desenvolver uma compreensão mais aprofundada dos desafios e das soluções associadas à manipulação de dados em memória dinâmica, além de aprimorar as habilidades de programação e resolução de problemas. Esse processo não só reforçou os fundamentos das estruturas de dados, mas também contribuiu para uma maior confiança na aplicação dessas técnicas em contextos reais.