IPCA INSTITUTO POLITÉCNICO DO CÁVADO E DO AVE

RELATÓRIO TRABALHO PRÁTICO - 1ª FASE

Diogo Alexandre Alves da Silva nº 31504

UC: Estruturas de Dados Avançadas

Professor: Luís Ferreira

Março, 2025

Índice

Índice de figuras

Introdução

Este trabalho foi desenvolvido no âmbito da disciplina de Estruturas de Dados Avançadas, onde o professor Luís Ferreira nos propôs um trabalho prático para avaliação.

Este trabalho prático teve como intuito, reforçar os conhecimentos adquiridos nas aulas principalmente nos tópicos de definição e manipulação de estruturas de dados dinâmicas, neste caso, uma lista.

Para além disso o trabalho aborda temas como, manipulação de ficheiros, modularização e a documentação do código.

Com isso em mente o trabalho que foi proposto pelo professor foi, considerar uma cidade com várias antenas em que cada antena é sintonizada em uma frequência específica indicada por um carácter, com isso deveria-mos fazer a gestão do mapa dessa cidade onde teríamos de ter cuidado com a interferência que as antenas poderiam causar.

Problema

O problema apresentado foi o seguinte:

Dado um mapa num ficheiro de texto (como demonstrado abaixo), o aluno deveria desenvolver uma biblioteca que fizesse as seguintes operações com o âmbito de aprofundar nas listas ligadas:

- Gestão de Antenas
  - Definição da estrutura, Adicionar e Remover
- Dedução automática do efeito nefastos
  - Definição da estrutura, Cálculo das posições e Inserção
- Leitura e Escritura em ficheiros
  - Ficheiros de texto
  - Ficheiros binários
- Listagem das Antenas e Efeitos causados

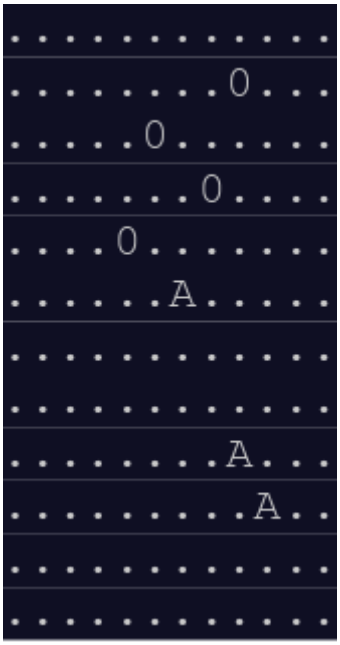

Imagem 1: Exemplo de mapa da cidade

O mapa é uma representação bidimensional onde cada casa representa diferentes tipos, por exemplo:

- '.' : Representa uma casa vazia no mapa.
- '#' : Representa uma área afetada por um efeito nefasto.

- Outro caractere: Representa uma antena, com uma frequência específica. Essas antenas podem causar efeitos nefastos se estiverem próximas umas das outras.

Quando duas antenas da mesma frequência encontram-se no mapa (Imagem 1)estas podem causar efeito nefasto. O efeito nefasto é causado numa localização onde duas antenas estão ao dobro da distância simetricamente ou seja se uma antena está na posição (3,3) e a outra está na posição (5,5) então vai haver efeito nefasto na posição (1,1) e (7,7) (se tiver dentro dos limites do mapa).

Essa lógica de propagação de efeitos nefastos é implementada automaticamente na biblioteca, sempre que uma nova antena é adicionada ou removida do mapa.

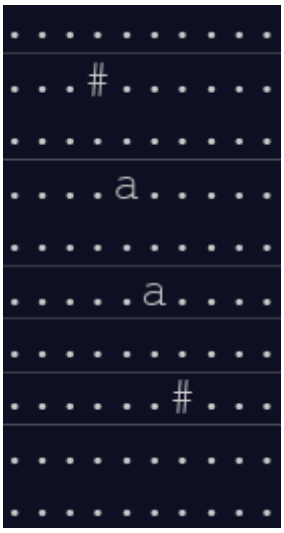

Imagem 2: Exemplo de efeito nefasto no mapa

Respositório GitHub:

Respositório GitHub

Desenvolvimento

Para iniciar este projeto tive que decidir como iria definir as regras/instruções de utilização do programa sendo elas:

- Apenas uma antena em cada posição
- Uma antena pode estar num local de efeito nefasto

Abordagem

A abordagem escolhida para a resolução do problema foi a implementação de Listas Ligadas Simples. Essa escolha deve-se à sua eficiência e simplicidade frente às exigências do projeto nas funções para trabalhar com estruturas de dados dinâmicas. O objetivo também foi aprofundar o conhecimento na definição, manipulação e uso prático de listas ligadas simples, explorando conceitos como a alocação dinâmica de memória.

Listas Ligadas Simples

Uma lista ligada é uma estrutura de dados dinâmica que armazena elementos (chamados de nós) de forma encadeada. Cada nó contém:

- Dados: Os valores guardados pela estrutura normal.
- Ponteiro: Um apontador para o próximo nó da lista.

Ao contrário dos arrays, as listas ligadas não precisam de memória contígua e permitem a inserção e a remoção de elementos de forma eficiente, já que basta ajustar os ponteiros sem realocar toda a estrutura.

Tipos comuns:

- Lista Ligada Simples: Cada nó aponta apenas para o próximo.
- Lista Duplamente Ligada: Cada nó aponta para o próximo e para o anterior.
- Lista Circular: O último nó aponta de volta para o primeiro, formando um ciclo.

Implementação

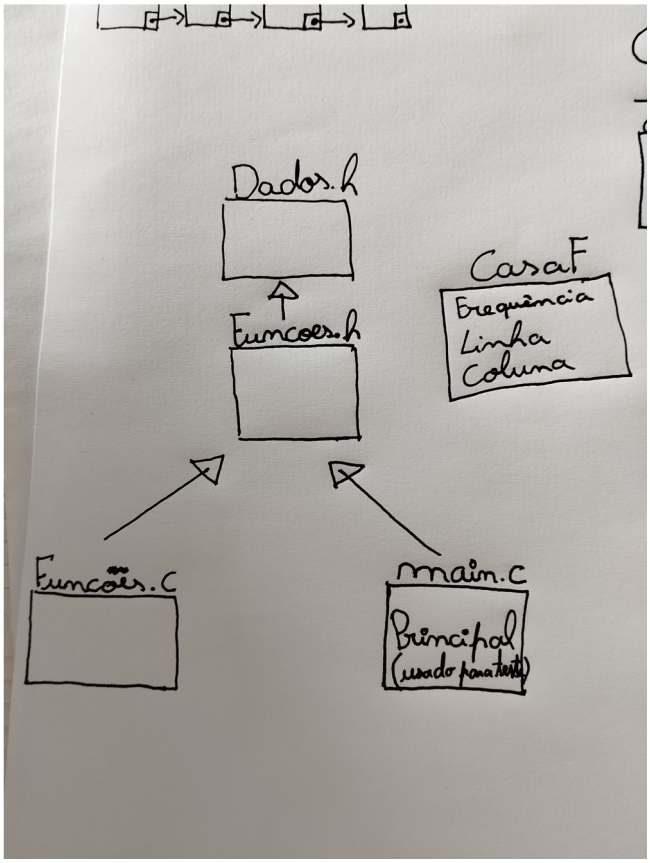

Imagem 3: Estrutura de ficheiros

dados.h

Neste ficheiro header estão declaradas as structs utilizadas ao longo do programa.

Structs

No início do trabalho teve de se planear a estrutura que iria guardar as informações do mapa (Ex: Imagem 1: Exemplo de mapa da cidade), ou seja, como a cada antena ou cada ponto do mapa seria guardado em que na primeira versão tinha esta feitura:

```
11  typedef struct Casa
12  {
13      char c;
14      int linha;
15      int coluna;
16      int efeitoNefasto;
17      struct Casa *prox;
18      struct Casa *ant;
19  }Casa;
```

Imagem 4: Struct Casa (Primeira versão)

Com esta estrutura seria mais que uma lista ligada simples e estaria a guardar os efeitos nefastos dentro das casas, pois nesta versão o mapa seria tudo guardado, por exemplo os '.' eram guardados na lista ligada, apenas não eram considerados antenas. Por isso o nome da struct ser *Casa*, pois iria ser cada casinha do mapa, ou seja num mapa de 5x3 a lista iria ter 15 "*Casas*" e isso não seria nem eficiente nem prático para casos futuros.

Para resolver esse e futuros problemas decidiu-se separar esta *struct* em 3 *structs* que não se mostrou apenas mais simples, como também se mostrou mais eficiente e modular, *struct Casa*, *struct CasaF* e *struct EfeitoNefasto* (Imagem 5: Structs - Versão Final).

```c
/**
 * @brief Struct que guarda as antenas
 *
 */
typedef struct Casa
{
    char c; //Frequência
    int linha; //linha no mapa
    int coluna; //coluna no mapa
    struct Casa *prox; //proxima antena na lista
}Casa;

/**
 * @brief Struct que ajuda a salvar as antenas
 *
 */
typedef struct CasaF
{
    char c; //Frequência
    int linha; //linha no mapa
    int coluna; //coluna no mapa
}CasaF;

/**
 * @brief Struct que guarda os efeitos nefastos
 *
 */
typedef struct EfeitoNefasto
{
    int linha; //linha no mapa
    int coluna; //coluna no mapa
    struct Casa *antenas[2];//Antenas que causam o efeito
    struct EfeitoNefasto* prox; // //proximo efeito nefasto na lista
}EfeitoNefasto;
```

Imagem 5: Structs - Versão Final

A *struct Casa* serve para guardar as Antenas do mapa; a *struct CasaF* serve para armazenar e ler antenas do ficheiro binário. Por fim, a *struct EfeitoNefasto* para poder guardar com precisão o efeito nefasto numa lista ligada que tem como extra o salvamento das antenas que provocaram a sua existência. Todas elas têm os campos *linha* e *coluna*, pois é isso que as localiza no mapa e nas structs criadas para as listas. Existe ainda nas *struct EfeitoNefasto* e *struct Casa*, um apontador (*prox)* para a próxima estrutura na memória, afim de suportar uma lista ligada.

funcoes.h e funcoes.c

Os arquivos funcoes.h e funcoes.c contêm os módulos que implementa as operações relacionadas ao de antenas e efeitos nefastos.

No ficheiro .h estão as declarações, ou seja, é aqui que os protótipos de todas as funções são apresentados. Essas declarações informam o compilador sobre o nome da função, os parâmetros e o

tipo de retorno, que permite a utilização do código noutras partes do programa. No ficheiro .c é onde a lógica das listas está implementada.

Criação de espaços na memória

Para a criação de espaços na memória a biblioteca tem duas funções (uma para cada tipo com lista ligada):

```
19  #pragma region Criacao
20  Casa *CriaCasa(char c,int linha,int coluna){···
32  EfeitoNefasto *CriaEfeitoNefasto(int linha,int coluna,Casa* x1, Casa* x2){···
45  #pragma endregion
```

Imagem 6: Cabeça das funções de alocação de memória

Estas funções recebem como parâmetros os valores necessários para preencher a memória alocada. Na Imagem 7: Exemplo de uma alocação de memória mostra o caso da função *CriaCasa*. Aqui vemos o exemplo de validação dos parâmetros (linhas 21-22), se é alocado um espaço na memória (linha 24) e devolve o apontador desse endereço de memória (linha 30).

```
19  #pragma region Criacao
20  Casa *CriaCasa(char c,int linha,int coluna){
21      if(coluna<1 || coluna>x || linha<1 || linha>y)return NULL; //validação localização
22      if(c=='.'|| c=='#')return NULL; //validação frequência
23      Casa* aux = (Casa*)malloc(sizeof(Casa)); //alocação de memória
24      if(aux!=NULL){//se alocou insere valores
25          aux->c=c;
26          aux->linha=linha;
27          aux->coluna=coluna;
28          aux->prox=NULL;
29      }
30      return aux;//retorna o apontador para a memório preenchida/NULL
31  }
```

Imagem 7: Exemplo de uma alocação de memória

Limpeza da memória

Como há alocações de memória, tem que haver uma limpeza no fim do programa correr então para isso foram definidas duas funções, uma para cada lista ligada (Imagem 8: Cabeça das funções de limpeza de memória).

```
47  Casa* LimpaMemoria(Casa* mapa){···
59  EfeitoNefasto* LimpaMemoriaEfeito(EfeitoNefasto* efeito){···
```

Imagem 8: Cabeça das funções de limpeza de memória

As duas funções agem de forma similar mas decidiu-se separar para o código ficar mais legível e para caso fosse necessário diferentes utilizações.

```
47  Casa* LimpaMemoria(Casa* mapa){
48      Casa *atual = mapa;//atual começa a apontar para a cabeça
49      Casa *proximo=NULL;//define proximo
50      while (atual)//enquanto diferente de NULL
51      {
52          proximo=atual->prox;//proximo vai ser o proximo ao atual (age como auxiliar)
53          free(atual);//libertamos o atual
54          atual = proximo;//atual vira o proximo da lista
55      }
56      mapa=NULL;
57      return mapa;//mapa retorna sempre NULL
58  }
```

Imagem 9: Limpa Memória da lista das antenas

Na Imagem 9: Limpa Memória da lista das antenas, podemos observar que a função recebe a cabeça da lista (linha 47). A função define duas variáveis auxiliares (linhas 48-49), vai percorrer a lista toda e enquanto vai passando para o próximo vai limpando a posição em que se encontra no momento (linhas 50-54). Como observamos, esta função retorna sempre *NULL* para feitos práticos (linhas 56-57).

Cálculo das posições

Para facilitar algumas operações que serão futuramente apresentadas neste documento deve-se falar antes de duas funções utilizadas amplamente ao longo deste código. Essas funções encontram-se no código definidas com a aspeto apresentado na Imagem 10: Cabeça das funções que calculam a posição da Casa/EfeitoNefasto passado.

```
73  #pragma region CálculoPosições
74  int Posicao(Casa* c){···
82  int PosicaoEfeitoNefasto(EfeitoNefasto* c){···
90  #pragma endregion
```

Imagem 10: Cabeça das funções que calculam a posição da Casa/EfeitoNefasto passado

Nem no caso do limpar memória nem nas funções de calcular a posição, nem em qualquer outra parte do código foi utilizado o método *Generic Pointer Casting*, ou seja, receber um ponteiro por parâmetro e fazer a função de uma forma genérica. Esta forma não foi utilizada pois um dos objetivos deste código era ser legível e de fácil utilização, assim fica de forma evidente que posição se está a calcular.

```
73  #pragma region CálculoPosições
74  int Posicao(Casa* c){
75      int posicao=0;//coloca posicao 0 por default (não se encontra no mapa)
76      if(c){//se casa existe
77          int linha= c->linha, coluna= c->coluna;// define linha e coluna conforme casa
78          posicao=((linha-1)*x)+coluna-1;//calcula a posicao
79      }
80      return posicao;//retorna a posicao 0 ou a calculada
81  }
```

*Imagem 11: Função que calcula a posição da casa no mapa (de 1 a n*m)*

Na Imagem 11: Função que calcula a posição da casa no mapa (de 1 a n*m), podemos observar o método de cálculo da posição de uma *Casa* no mapa. A função apenas verifica se o valor passado por parâmetro é válido e se for faz o cálculo da linha 77 (Imagem 12: Desenhos necessários para chegar ao cálculo da posição). Por fim retorna ou o valor calculado ou 0 (linha 80).

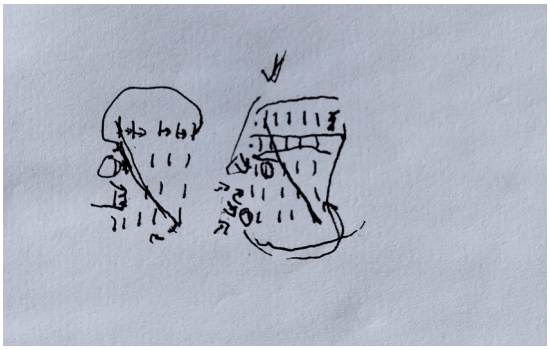

Imagem 12: Desenhos necessários para chegar ao cálculo da posição

Adicionar

Para a opção de adicionar, foram feitas duas funções, *AdicionaCasa e AdicionaCasaEfeitoNefasto* (Imagem 13: Cabeça das funções de Adicionar linhas 94 e 131 respetictivamente), ambas têm o mesmo comportamento, com as suas devidas funções a serem utilizadas mas será explicada a função *AdicionaCasa* (linha 94) pois tem uma “exclusividade”.

```
92  #pragma region Adicionar
93
94  Casa* AdicionaCasa(Casa *mapa,Casa* n,EfeitoNefasto** cabeca){···
131 EfeitoNefasto* AdicionaCasaEfeitoNefasto(EfeitoNefasto* cabeca,EfeitoNefasto* n){···
166
167 #pragma endregion
```

Imagem 13: Cabeça das funções de Adicionar

```
94  Casa* AdicionaCasa(Casa *mapa,Casa* n,EfeitoNefasto** cabeca){
95      if(!n) return mapa;//se Casa passada é nulo retorna mapa normal
96      if(!mapa){//se mapa é nulo, cria mapa
97          mapa=n;
98          return mapa;
99      }
100     Casa *atual=mapa;
101     while (atual)
102     {
103         if(Posicao(n)==Posicao(atual)){//se a posição da inserção é igual a uma antena já no mapa
104             printf("Erro, já existe antena na posição (%d,%d)\n",n->linha,n->coluna);
105             free(n);
106             return mapa;
107         }
108         if(Posicao(n)<Posicao(atual)){
109             //adicionar inicio
110             n->prox=atual;
111             mapa=n;
112             *cabeca=CriaListaEfeitoEfeitoNefasto(*cabeca,mapa);
113             return mapa;
114         }else if(atual->prox && Posicao(n)<Posicao(atual->prox)){
115             //adicionar ao meio
116             n->prox=atual->prox;
117             atual->prox=n;
118             *cabeca=CriaListaEfeitoEfeitoNefasto(*cabeca,mapa);
119             return mapa;
120         }
121         if(atual->prox==NULL){
122             //adicionar ao fim
123             atual->prox=n;
124             *cabeca=CriaListaEfeitoEfeitoNefasto(*cabeca,mapa);
125             return mapa;
126         }
127         atual=atual->prox;
128     }
129     return mapa;
130 }
```

Imagem 14: Função AdicionaCasa em detalhe

A função AdicionaCasa(Imagem 14: Função AdicionaCasa em detalhe) recebe como parâmetros, a cabeça do mapa, a casa que se vai adicionar e um apontador para o apontador da cabeça (linha 94).

Esta escolha de adicionar este último parâmetro veio de não ter que haver a necessidade de criar uma *Sruct* para armazenar as duas cabeças das listas para devolver a quem estiver a usar a função,

com um duplo apontador pode-se modificar a lista dentro da própria função sem ter que retornar nada.

A função começa com umas verificações (linhas 95-99), depois se tudo estiver correto percorre a lista e quando encontrar a posição a que pertence adiciona a antena à lista de antenas, vai criar uma lista de efeitos nova () e coloca essa lista na cabeça passada por parâmetro(Ex: linhas114-120).

Funções que criam e adicionam

Estas funções foram criadas com o único intuito de simplificar código, o que elas fazem é resumir passos para quem estiver a utilizar a biblioteca.

```
169   #pragma region CriaAdiciona
170 > Casa *CriaAdiciona(Casa *mapa,char c, int linha, int coluna,EfeitoNefasto* cabeca){···
173 > EfeitoNefasto *CriaAdicionaEfeito(EfeitoNefasto *cabeca, int linha, int coluna, Casa* x1,Casa* x2){···
176
177   #pragma endregion
```

Imagem 15: Funções que criam e adicionam

Estas são duas funções que apenas juntam as funções já vistas acima (Criação de espaços na memória e Adicionar)

Remover

Para as funções de remover, para o efeito nefasto e para a antena segue também uma estrutura de código semelhante, as duas funções percorrem a sua lista e verifica a linha e a coluna que tem que ser removida. Mas na função para remover o efeito nefasto (Imagem 16: Função para remover o efeito nefasto) há uma ligeira diferença.

```c
EfeitoNefasto *RemoverNefasto(EfeitoNefasto* cabeca, int linha, int coluna){
    EfeitoNefasto *atual=cabeca;
    if((atual) && (((atual->antenas[0]->coluna==coluna) && (atual->antenas[0]->linha==linha)) ||
     ((atual->antenas[1]->coluna==coluna) && (atual->antenas[1]->linha==linha)))){
        //verifica se as antenas que o causaram estão na linha e coluna da casa que se vai remover
        EfeitoNefasto *temp = atual;
        atual = atual->prox;
        free(temp);
        cabeca = atual;
    }
    while (atual)
    {
        if((atual->prox) && (((atual->prox->antenas[0]->coluna==coluna) && (atual->prox->antenas[0]->linha==linha)) || ((atual->prox->antenas[1]->linha==linha) && (atual->prox->antenas[1]->coluna==coluna)))){
            //verifica se existe proximo e se existir verificar se as antenas que o causaram estão na linha e coluna da casa que se vai remover
            EfeitoNefasto *temp = atual->prox;
            atual->prox = temp->prox;
            free(temp);
            cabeca=RemoverNefasto(cabeca,linha,coluna);
        } else if((!atual->prox) && (((atual->antenas[0]->coluna==coluna) && (atual->antenas[0]->linha==linha)) ||
        ((atual->antenas[1]->coluna==coluna) && (atual->antenas[1]->linha==linha)))){
            //é o ultimo efeito nefasto
            if(cabeca==atual){
                cabeca=NULL;
                free(atual);
                return cabeca;
            }
            free(atual);
            cabeca= RemoverNefasto(cabeca,linha,coluna);
        }
        atual=atual->prox;
    }
    return cabeca;

}

#pragma endregion
```

Imagem 16: Função para remover o efeito nefasto

Como se observa na Imagem 16: Função para remover o efeito nefasto, a função verifica se é o primeiro da lista a ser removido, se é remove, mas não para continua e entra no ciclo para percorrer a lista. Dentro da lista verifica se as antenas causadoras do efeito nefasto possuem a linha e coluna passadas por parâmetro, se sim remove e chama a função de novo para percorrer a lista de novo para voltar a remover.

Função *CriaListaEfeitoNefasto*

O efeito nefasto, como já referido anteriormente, é feito em forma simétrica em ambas as direções das antenas (em frente na segunda antena e para trás na primeira) para isso foi criada esta função, para percorrer a lista de antenas e criar o efeito nefasto das respetivas antenas.

```
254 EfeitoNefasto* CriaListaEfeitoEfeitoNefasto(EfeitoNefasto* cabeca, Casa *mapa){
255     cabeca=LimpaMemoriaEfeito(cabeca); //dá reset ao efeito nefasto
256     Casa* atual= mapa,*prox = atual->prox;
257     int difC, difL;
258     while(atual){
259         prox=atual->prox;
260         while (prox)
261         {
262             if(prox->c==atual->c){
263                 difL=prox->linha-atual->linha;
264                 difC=abs(atual->coluna-prox->coluna);
265                 if(atual->coluna<prox->coluna){
266                     if(prox->coluna+difC<=x && prox->linha+difL<=y){//efeito nefasto para frente
267                         cabeca=CriaAdicionaEfeito(cabeca,prox->linha+difL,prox->coluna+difC,atual,prox);
268                     }
269                     if(atual->coluna-difC>0 && atual->linha-difL>0){//efeito nefasto para trás
270                         cabeca=CriaAdicionaEfeito(cabeca,atual->linha-difL,atual->coluna-difC,atual,prox);
271                     }
272                 }else{
273                     if(prox->coluna-difC>0 && prox->linha+difL<=y){//efeito nefasto para frente
274                         cabeca=CriaAdicionaEfeito(cabeca,prox->linha+difL,prox->coluna-difC,atual,prox);
275                     }
276                     if(atual->coluna+difC<=x && atual->linha-difL>0){//efeito nefasto para trás
277                         cabeca=CriaAdicionaEfeito(cabeca,atual->linha-difL,atual->coluna+difC,atual,prox);
278                     }
279                 }
280 
281             }
282             prox=prox->prox;
283         }
284         atual=atual->prox;
285     }
286     return cabeca;
287 }
```

Imagem 17: Função CriaListaEfeito

Nesta função, ele percorre a lista das antenas enquanto percorre também o próximo.

Para calcular corretamente teve-se de inserir as 5 condições encontradas na Imagem 17: Função CriaListaEfeito (linhas 265-276) que se corresponderem adiciona um efeito nefasto na linha e coluna calculadas e verificadas nas condições.

Funções de printar

```
244 void MostraListaNovo(Casa *mapa,EfeitoNefasto* cabeca){
245     Casa* atual=mapa;
246     for(int i=1;i<=y;i++){
247         for (int j = 1; j <= x; j++)
248         {
249             if(atual && atual->linha==i && atual->coluna==j){
250                 if(ExisteEfeitoEfeitoNefasto(cabeca,i,j)){
251                     printf("\033[0;33m%c\033[0m",atual->c);
252                 }else{
253                     printf("%c",atual->c);
254                 }
255                 atual=atual->prox;
256             }else{
257                 if(ExisteEfeitoEfeitoNefasto(cabeca,i,j)){
258                     printf("#");
259                 }
260                 else{
261                     printf("%c",vazio);
262                 }
263             }
264             if(j==x){
265                 printf("\n");
266             }
267 
268         }
269 
270     }
271 }
272 void MostraListaCasas(Casa *mapa,EfeitoNefasto* cabeca){
273     Casa* atual=mapa;
274     EfeitoNefasto*  efeito=cabeca;
275     printf("Frequência | Linha | Coluna |\n");
276     while (atual)
277     {
278         printf("     %c     |  %d   |   %d   |\n",atual->c,atual->linha,atual->coluna);
279         atual=atual->prox;
280     }
281     while (efeito)
282     {
283         printf("     %c     |  %d   |   %d   | Casa 1:%c(%d,%d)| Casa 2:%c(%d,%d) \n",'#',efeito->linha,efeito-
284 
285         efeito=efeito->prox;
286     }
287 }
288 #pragma endregion
```

Imagem 18: Funções de print

Foram definidas duas funções de *print (Imagem 18: Funções de print), a* primeira (linha 244) escreve na consola um mapa igual ao mapa no ficheiro de texto mas que *printa* com efeitos nefastos e antenas que estejas com interferência. Para isso encontra-mos uma condição (linha 250) que se houver efeito nefasto e tiver uma antena no local(linha 249) escreve com a frequência com a cor amarela, se não tiver ruído escreve a frequência normal, se não tiver frequência ou escreve ruído ou escreve vazio (‘.’).

Funções de Guardar Listas

Para as funções de guardar as listas, o código tem dois tipos:

1. Texto
2. Binário

Estas funções utilizam duas variáveis definidas globalmente, o y e o x que são as linhas e as colunas, respetivamente.

Nas funções de txt temos as seguintes funções

```
290  #pragma region LerEscreverFicheiros
291
292  Casa* CriaMapaCasas(char* nome,Casa *mapa,EfeitoNefasto** hE){
293      FILE* ficheiro = fopen(nome, "r");
294      if (ficheiro == NULL) {//verifica se abriu corretamente
295          printf("Erro ao abrir o arquivo %s\n", nome);
296          return NULL;
297      }
298      if(mapa!=NULL){
299          mapa=LimpaMemoria(mapa);
300      }//se mapa não for nulo limpa a lista
301      int c=0;
302      while ((c = fgetc(ficheiro)) != EOF) {
303          if (c == '\n') {
304              y++;
305              x=0;
306          } else {
307              x++;
308              if(c!=vazio){//Se o caractere não for uma casa vazia, guarda
309                  mapa=CriaAdiciona(mapa,c,y,x,hE);
310              }
311          }
312      }//le todos os caracteres um a um
313      fclose(ficheiro);
314      return mapa;
315  }
316  void criaMapaFicheiro(Casa *mapa){
317      FILE* ficheiro;
318      Casa* atual=mapa;
319      ficheiro = fopen("mapa.txt", "w");
320      if (ficheiro == NULL) {//verifica se abriu corretamente
321          printf("Erro ao abrir o ficheiro.\n");
322          return;
323      }
324      for(int i=1;i<=y;i++){
325          for (int j = 1; j <= x; j++)
326          {
327              if(atual && atual->linha==i && atual->coluna==j){//se tiver frequencia
328                  fprintf(ficheiro,"%c",atual->c);
329                  atual=atual->prox;
330              }else{
331                  fprintf(ficheiro, "%c",vazio);//se nao tiver nada
332              }
333              if(j==x && i!=y){//se chegar a ultima coluna
334                  fprintf(ficheiro,"\n");
335              }
336
337          }
338
339      }
340
341      fclose(ficheiro);
342  }
```

Imagem 19: Funções de guardar e ler em txt

Na função para ler o mapa do ficheiro tem a função na linha 292 (*CriaMapaCasa)* em que ele tenta abrir o ficheiro, se abrir e a cabeça da lista não for nula limpa a memória. Depois de limpar a memória ele lê todos os caracteres do ficheiro. Se não chegar ao fim do ficheiro ele verifica qual é o caratere, se for '\n' passa de linha e *reseta* as colunas (linha 303-305). Depois se não for uma casa vazia ele guarda a antena na lista do tipo *Casa.*

Na última função ele tem um funcionamento equivalente mas para escrever no ficheiro de texto, para ver se coloca a frequência ele compara as variáveis dos ciclos com as da antena que tem na lista para ver se é no local correto a inserir.

Nos ficheiros em binário temos outras duas funções, *LerListaFicheiro e EscreveListaFicheiro*

```
344 Casa* LerListaFicheiro(Casa *mapa,EfeitoNefasto** cabeca){
345     mapa=LimpaMemoria(mapa);
346     FILE* ficheiro= fopen("ListaCasas.bin","rb");
347     CasaF aux;
348     if (ficheiro == NULL) {//verifica se abriu corretamente
349         printf("Erro ao abrir o ficheiro.\n");
350         return NULL ;
351     }
352     while (fread(&aux,sizeof(aux),1,ficheiro)==1)
353     {
354         mapa= CriaAdiciona(mapa,aux.c,aux.linha,aux.coluna,cabeca);
355     }
356     fclose(ficheiro);
357     *cabeca = CriaListaEfeitoEfeitoNefasto(*cabeca, mapa);
358     return mapa;
359 }
360 Casa* EscreverListaFicheiro(Casa *mapa){
361     FILE* ficheiro= fopen("ListaCasas.bin","wb");
362     Casa* atual= mapa;
363     CasaF aux;
364     if (ficheiro == NULL) {//verifica se abriu corretamente
365         printf("Erro ao abrir o ficheiro.\n");
366         return NULL;
367     }
368     while (atual)
369     {
370         aux.c=atual->c;
371         aux.coluna =atual->coluna;
372         aux.linha = atual->linha;
373         fwrite(&aux,sizeof(aux),1,ficheiro);
374         atual=atual->prox;
375     }
376     fclose(ficheiro);
377     return mapa;
378 }
379 #pragma endregion
```

Imagem 20: Funções de ler e escrever(Ficheiros binários)

Estas funções tem um comportamento parecido com as funções de texto (Imagem 19: Funções de guardar e ler em txt) mas ao invés de utilizar a forma tabular, é tudo escrito/lido seguidamente e a segunda maior diferença é a utilização da *struct CasaF* para servir de intermediário entre a leitura e escrita dos ficheiros. Para isso utilizou-se o *fread* e o *fwrite* (linhas 352 e 373, respetivamente).

Função para verificar efeitos nefastos

```
382 int ExisteEfeitoEfeitoNefasto(EfeitoNefasto *cabeca, int linha, int coluna){
383     EfeitoNefasto* atual= cabeca;
384     while(atual){
385         if(atual->linha==linha && atual->coluna == coluna){
386             //verifica se existe efeito nefasto na linha e coluna indicada
387             return 1;
388         }
389         atual=atual->prox;
390     }
391     return 0;
392 }
```

Imagem 21: Função que verifica o efeito nefasto numa posição

Esta função foi utilizada por várias acima (Ex: Imagem 18: Funções de print) e serve para verificar se existe um efeito nefasto na posição (linha,coluna) (passados por parâmetro) o que facilita na reutilização de código.

Funções adicionais

```
426 }
427 EfeitoNefasto* AntenaCausaNefasto(EfeitoNefasto *cabeca, Casa* c){···
438 Casa* AdicionaCasaSemSobreposicao(Casa *mapa,Casa* n,EfeitoNefasto* cabeca){···
451
452 Casa* AdicionaCasaSemCausarNefasto(Casa *mapa,Casa* n,EfeitoNefasto* cabeca){···
```

Imagem 22: Funções adicionais (AntenaCausaEfeito, AdicionaCasaSemSobreposicao e AdicionaCasaSemCausarNefasto)

Estas três funções, foram feitas como um bónus que seria para verificar se uma casa ao ser inserida causará efeito nefasto (linha 427), outra para adicionar uma casa apenas se não tivesse um efeito nefasto (438) e por fim uma que só adicionava a casa se não causasse o efeito nefasto ,ou seja utilizando a função da linha 427 (linha 452).

main.c

O ficheiro *main.c* foi utilizado como um ficheiro de testes para analisar e verificar se as funções apresentadas no documento até agora estavam todas bem e todas a funcionar corretamente.

Conclusão

Este trabalho permitiu aprofundar os conhecimentos em estruturas de dados dinâmicas, principalmente na implementação e manipulação de listas ligadas simples. A abordagem escolhida não foi baseada apenas na eficiência e simplicidade, mas também para aprofundar o conhecimento no tipo de estrutura.

A implementação da biblioteca envolveu operações fundamentais, como inserção, remoção e listagem de antenas, além do cálculo e gestão dos efeitos nefastos no mapa. A integração com leitura e escrita de ficheiros também garantiu a persistência dos dados, permitindo maior controle sobre a estrutura do sistema, o que aumentou a dificuldade do trabalhou mas que aumentou a sua dinâmica.

Por fim, a modularização e a documentação com Doxygen ajudaram a tornar o código mais organizado e compreensível. O desenvolvimento deste projeto reforçou a importância da escolha adequada de estruturas de dados para cada problema e demonstrou como conceitos teóricos podem ser aplicados de forma prática.