

**Trabalho Prático 2**

**Word Ladder**

**Licenciatura em Engenharia Informática**

**Relatório**

**Turma P8**

**Professor:**

Joaquim Madeira ([*jmadeira@ua.pt*](mailto:jmadeira@ua.pt))

Tomás Oliveira da Silva ([*tos@ua.pt*](mailto:tos@ua.pt))

**Realizado por:**

Diogo Falcão Nº108712 – **50%**

José Gameiro Nº108840 – **50%**

08/01/2023

***Índice:***

[***1.*** ***Introdução*** 3](#_Toc124024432)

[***2.*** ***Desenvolvimento*** 4](#_Toc124024433)

[**2.1.** **Estruturas** 4](#_Toc124024434)

[**2.2.** **Funções** 6](#_Toc124024435)

[2.2.1. allocate\_adjency\_node () 6](#_Toc124024436)

[2.2.2. free\_adjency\_node () 6](#_Toc124024437)

[2.2.3. allocate\_hash\_table\_node () 7](#_Toc124024438)

[2.2.4. free\_hash\_table\_node () 7](#_Toc124024439)

[2.2.5. crc32 () 8](#_Toc124024440)

[***4.*** ***Conclusão*** 9](#_Toc124024441)

[***5.*** ***Webgrafia*** 9](#_Toc124024442)

# ***Introdução***

No âmbito da cadeira de Algoritmos e Estruturas de Dados (AED), foi-nos proposto realizar um projeto que consiste na criação de uma *Word Ladder*, ou seja, uma *escada de palavras*, com base em conceitos que foram lecionados nas aulas teóricas e práticas, que são, implementação de métodos para o desenvolvimento de uma *Hash Table*, utilização de *Linked Lists* e propriedades e teoremas de grafos.

Um *Word Ladder* ou uma *escada de palavras* consiste em, dentro de um conjunto constituído por várias palavras diferentes, selecionar duas e ser possível partir de uma palavra e chegar à outra alterando apenas uma letra. Tal como referido as palavras encontram-se todas num conjunto que é uma *Hash Table*. Uma *Hash Table* é uma estrutura de dados que usa funções hash para armazenar e recuperar dados de forma rápida, as funções são usadas para converter os dados numa chave de hash, que é então usada para armazenar os dados na tabela de hash. É possível ocorrerem colisões de hash, em que ao utilizar a mesma função de hash em uma ou mais chaves, é gerado o mesmo valor de hash para cada chave. Quando isto acontece é criada uma *Linked List* para armazenar as diferentes chaves com o mesmo valor de hash. Uma *Linked List* é, também, uma estrutura de dados de listagem linear que é composta por nós, cada nó contém um campo para armazenar dados e um ponteiro para o próximo nó presente na lista, à exceção do último nó, que aponta para *NULL*, indicando o final da lista.

Este relatório irá demonstrar e explicar as funções criadas para a implementação da *Hash Table*, o raciocínio que tivemos para a sua implementação bem como os testes que fizemos para perceber se a nossa implementação estava correta.

# ***Desenvolvimento***

Para a implementação da *Word Ladder*, foi-nos disponibilizado um conjunto de ficheiros, no qual se encontrava um script, designado por *word\_ladder.c*, com várias funções incompletas para a implementação de uma *Hash Table* e de um grafo em C, também se encontram presentes cinco ficheiros de texto que foram utilizados para testar a nossa *Hash Table*:

* ***wordlist-big-latest.txt*:** contém um número muito elevado de palavras com diferentes tamanhos;
* ***wordlist-four-letters.txt*:** contém várias palavras com tamanho de quatro letras, que foram retiradas do ficheiro ***wordlist-big-latest.txt***;
* ***wordlist-five-letters.txt*:** contém várias palavras com tamanho de cinco letras, que foram retiradas do ficheiro ***wordlist-big-latest.txt***;
* ***wordlist-six-letters.txt*:** contém várias palavras com tamanho de seis letras, que foram retiradas do ficheiro ***wordlist-big-latest.txt***;
* ***teste.txt*:** foi um ficheiro criado por nós, que contém um número de palavras reduzido, usado maioritariamente para nós testarmos a nossa implementação.

A partir deste ponto do relatório iremos explicar o algum conteúdo que se encontra já implementado no ficheiro word\_ladder.c, bem como o código que nós adicionámos às funções que se encontravam incompletas.

## **Estruturas**

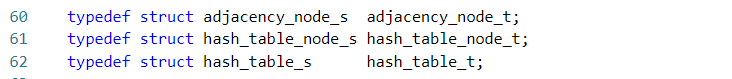


Fig.1 – Estruturas criadas

Foram criadas três estruturas para a implementação da *Hash Table* e do grafo:

* ***adjency\_node\_s* ou *adjency\_node\_t***: A primeira estrutura denomina-se por *adjency\_node\_s*, que representa um nó de uma lista de adjacência. Esta estrutura é utilizada para a representação do grafo, em que cada nó da lista armazena um vértice e um conjunto de vértices adjacentes a esse vértice. Tem como atributos a variável *next* que é um ponteiro e é do tipo de ***adjency\_node\_t*** e a variável *vertex* que também é um ponteiro e é do tipo ***hash\_table\_node\_t*.**

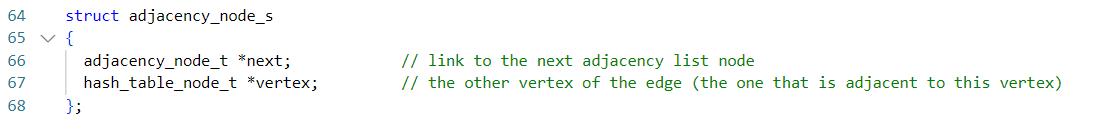


Fig.2 – Estrutura *adjency\_node\_s*

* ***hash\_table\_node\_s* ou *hash\_table\_node\_t***: esta estrutura representa um nó da *Hash Table*. Tem como atributos:

- ***word*:** será a palavra que irá ficar guardada no nó, esta é um vetor de caracteres que podem ter como tamanho máximo 32 caracteres;

- ***next***: é um ponteiro para outro nó da *Hash Table*, que é usado para ligar os nós através de uma *Linked List* dentro da *Hash Table*;

- ***head***: é um ponteiro para o primeiro nó da lista de adjacência;

- ***visited***: uma variável do tipo inteiro que indica se o vértice foi visitado ou não;

- ***previous***: representa um ponteiro para o nó anterior, que irá ser utilizado para a função breadh\_first\_search;

- ***representative***: é um ponteiro para o nó representante da componente conexa ao qual esse vértice pertence.

- ***number\_of\_vertices***: uma variável do tipo integer que indica o número de vértices da componente conexa à qual o vértice pertence.

- ***number\_of\_edges***: uma variável do tipo inteiro que indica o número de arestas da componente conexa à qual pertence o vértice.

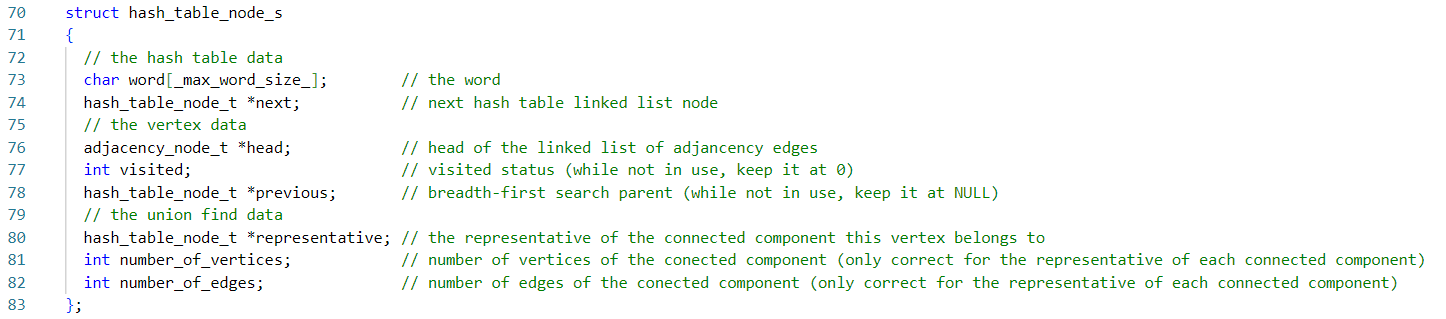


Fig.3 – Estrutura *hash\_table\_node\_s*

* ***hash\_table\_s* ou *hash\_table\_t*:** esta estrutura é utilizada para representar a *Hash Table*, e tem como atributos:

**- *hash\_table\_size*:** uma variável do tipo integer que indica o tamanho da *Hash Table*;

**- *number\_of\_entries*:** é uma variável do tipo integer que indica o número de entradas na *Hash Table*;

**- *heads*:** representa um ponteiro que aponta para um vetor do tipo *hash\_table\_node\_t*. Ao se inserir um elemento neste vetor ele irá ser inserido no início de uma lista ligada.

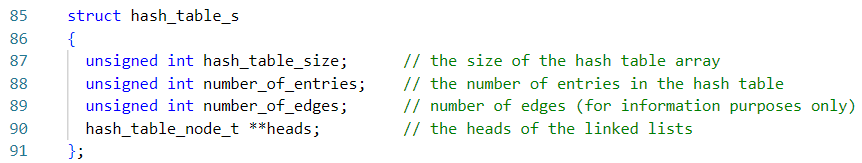
****

Fig.4 – Estrutura *hash\_table\_s*

## **Funções**

Neste ponto iremos explicar algumas funções que nos foram fornecidas e as funções que nós completámos

### *allocate\_adjency\_node ()*

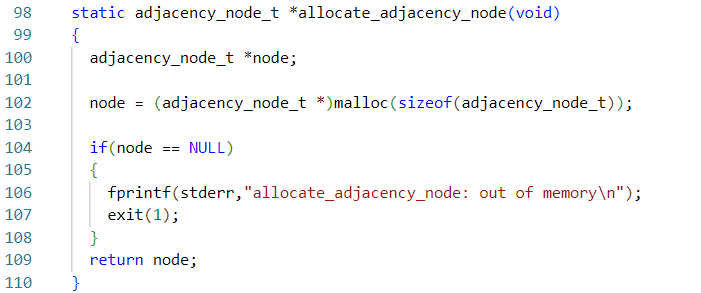


Fig.5 – Função *allocate\_adjency\_node ()*

Esta função tem como objetivo alocar de forma dinâmica um nó de uma lista de adjacência. Ela começa por declarar a variável *node* e, usando a função *malloc*, aloca um bloco de memória do tamanho especificado pelo tamanho da estrutura *adjency\_node\_t*. Caso a função *malloc* não funcione, ou seja, se a variável *node* for igual a NULL, isto significa que não existe memória suficiente para a alocação e a função imprime uma mensagem de erro. Caso a função malloc funcione, a função retorna um ponteiro para o bloco de memória alocado.

Esta função é simples e bastante útil pois simplifica o processo de alocação de nós da lista de adjacência que virá a ser necessário utilizar noutras funções.

### *free\_adjency\_node ()*

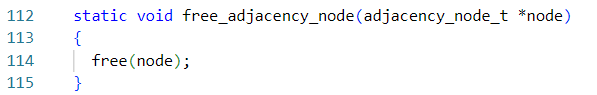


Fig.6 – Função *allocate\_adjency\_node ()*

A função *free\_adjency\_node* é uma função bastante simples pois apresenta apenas uma única instrução que consiste em utilizar a função *free ()* para libertar o espaço de memória alocado para o nó pretendido (o argumento de entrada da função). É importante libertar a memória alocada quando esta já não for necessário, para evitar vazamentos de memória.

### *allocate\_hash\_table\_node ()*

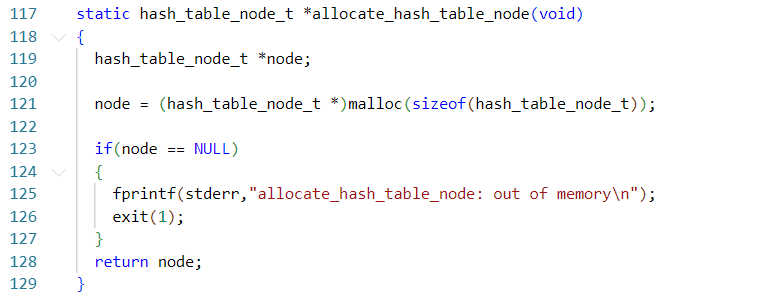


Fig.7 – Função *allocate\_hash\_table\_node ()*

A função *allocate\_hash\_table\_node* é muito semelhante *allocate\_adjency\_node ()*, pois ambas têm o mesmo objetivo que é alocar memória para um nó, só que no caso desta função o bloco de memória alocado irá ser atribuído a um nó da *Hash Table*.

### *free\_hash\_table\_node ()*

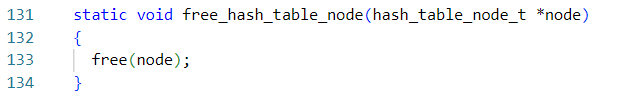


Fig.8 – Função *free\_hash\_table\_node ()*

Esta função é bastante semelhante à função *free\_adjency\_node ()*, pois ambas apresentam apenas uma instrução que tem o mesmo objetivo para libertar o espaço alocado, só que nesta função o espaço libertado será o alocado para um nó da *Hash Table*.

### *hash\_table\_create ()*

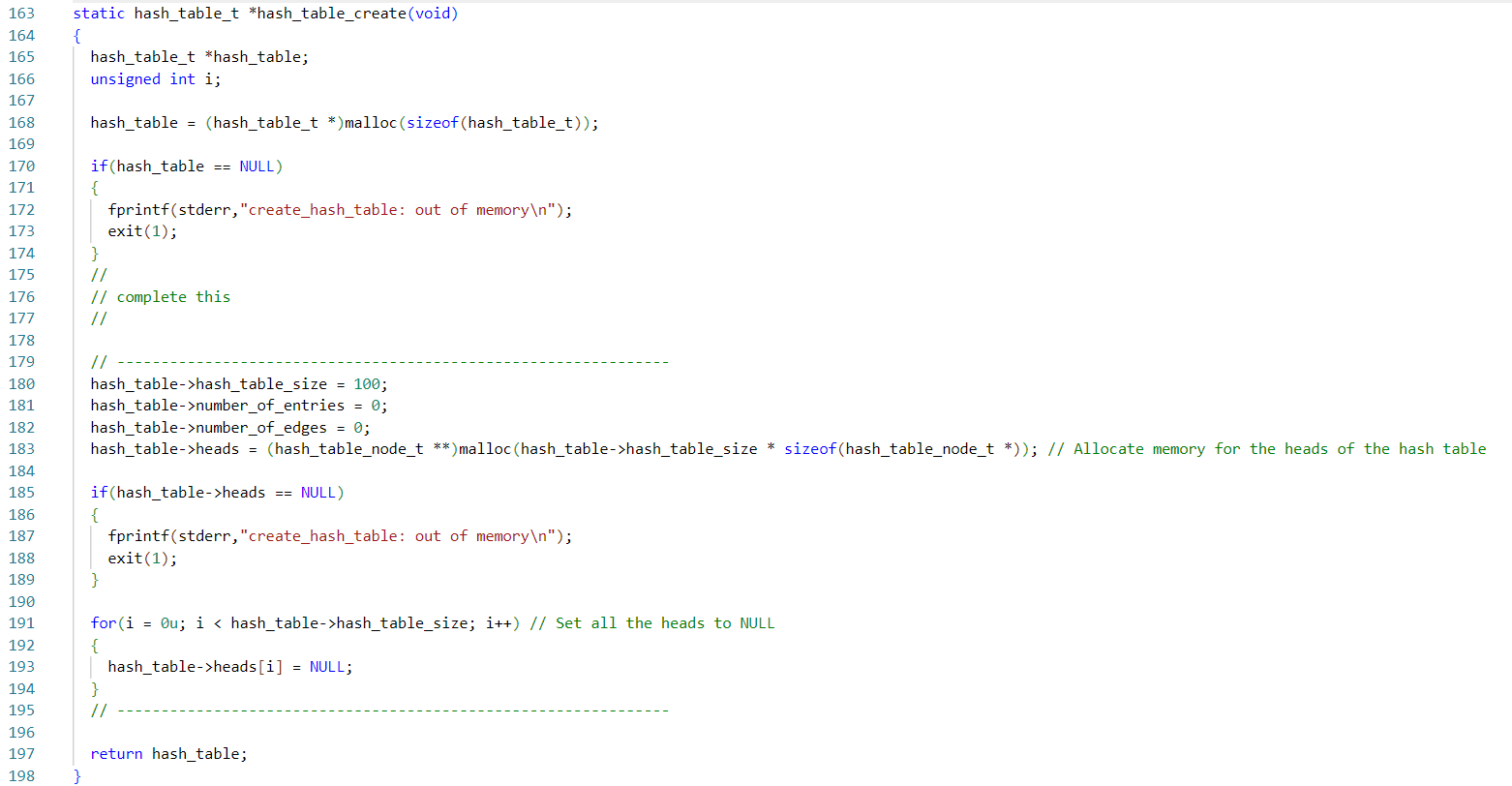
****

Fig.9 – Função *hash\_table\_create ()*

Esta função serve para inicializar uma *Hash Table*. Como primeiro passo é alocar um bloco de memória, através da função *malloc*, para a *Hash Table*, com tamanho igual ao da estrutura *hash\_table\_t*. Caso a alocação de memória não funcione é imprimido uma mensagem de erro e programa termina, caso contrário é atribuído como tamanho inicial da *Hash Table* cem, são inicializados o número de entradas e de arestas da *Hash Table* e é alocado espaço para o *array heads*, novamente com a função *malloc*, e verifica se a operação foi bem-sucedida. Como útlimo passo são inicializados todos os elementos que se encontram no array heads a NULL e termina a função com o retorno da *Hash Table* criada.

### *hash\_table\_free ()*

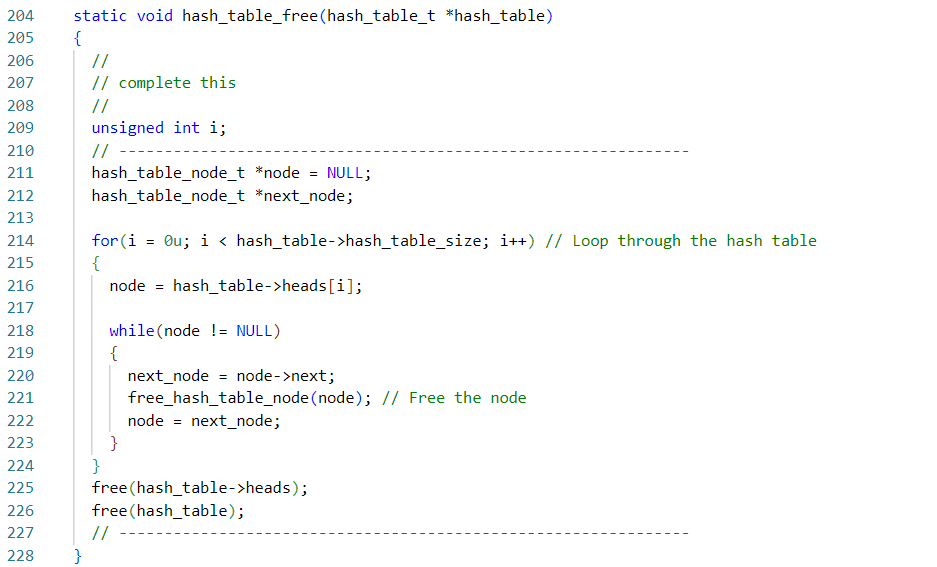
****

Fig.9 – Função *hash\_table\_free ()*

A função *hash\_table\_free ()* tem como objetivo libertar todo o espaço que foi alocado para a *Hash Table* e para os seus elementos.

O primeiro passo desta função é libertar os elementos do *array heads*, ou seja, o *array* que contém os nós da *Hash* *Table*, isto é feito através do ciclo for na linha 214, em que para cada elemento do *array*, é libertado o espaço que está no nó, avança para o próximo nó e termina quando a lista estiver vazia.

Depois de ser libertado o espaço de todos os nós, é também libertado o espaço de memória alocado para o *array heads* e para a estrutura *hash\_table\_t*.

Tal como as funções *free\_hash\_table\_node ()* e *free\_adjency\_node ()*, esta função previne a ocorrência de *memory leaks*.

### *hash\_table\_grow ()*

# ***Conclusão***

Com a realização deste trabalho concluímos que existem várias maneiras de abordar o problema proposto. Estas soluções utilizam métodos recursivos e iterativos. Constatamos que os métodos iterativos são mais rápidos e eficientes que os recursivos através da análise de gráficos de execução e dos ficheiros gerados pelo programa.

Observámos que, na criação de uma solução usando programação dinâmica, esta foi a melhor solução até à data visto que o gráfico do seu tempo de execução e o esforço obtido foram os menores.