

**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*** 2](#_Toc124013861)

[***2.*** ***Desenvolvimento*** 3](#_Toc124013862)

[**2.1.** **Estruturas** 4](#_Toc124013863)

[***4.*** ***Conclusão*** 8](#_Toc124013864)

[***5.*** ***Webgrafia*** 8](#_Toc124013865)

# ***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 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:

* **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**

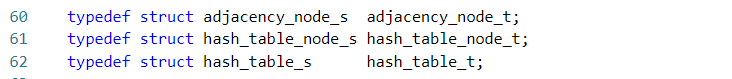


Fig.1 – Estruturas criadas

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.**

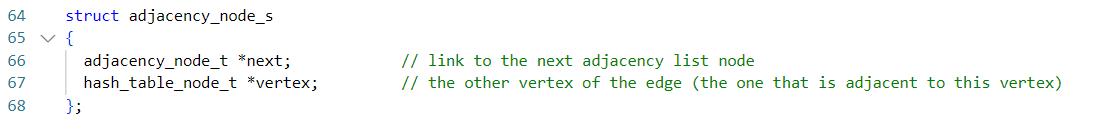


Fig.2 – Estrutura adjency\_node\_s

* **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 tabela de hash, 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.

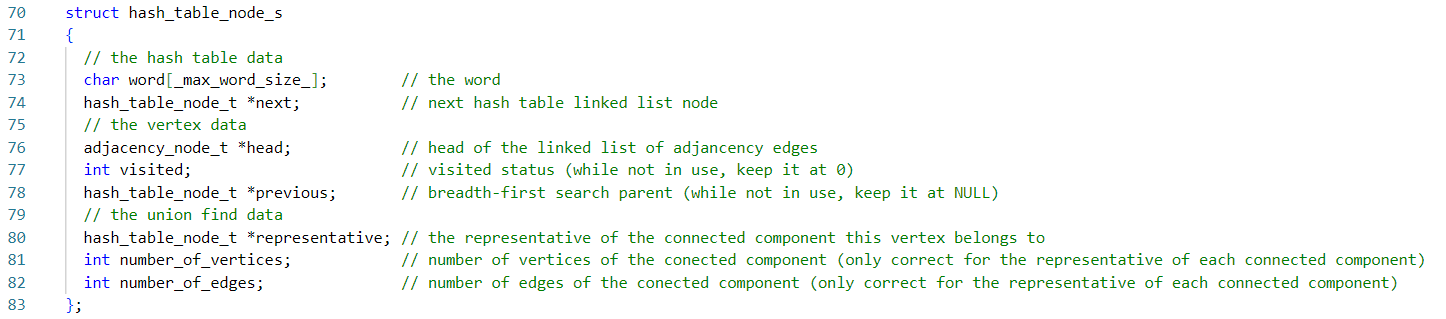


Fig.3 – Estrutura hash\_table\_node\_s

* **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.

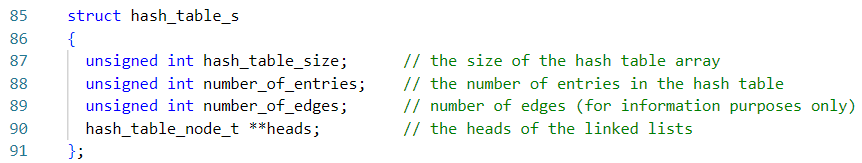
****

Fig.4 – Estrutura hash\_table\_s

# ***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.

# ***Webgrafia***

GEEKS (2022), Branch and Bound Algorithm, [visitado em 5 de dezembro de 2022]: <https://www.geeksforgeeks.org/branch-and-bound-algorithm/>

GEEKS (2022), Back Traking Algortithm, [visitado em 5 de dezembro de 2022]: <https://www.geeksforgeeks.org/backtracking-algorithms/>

GATWIRI (2022), Dynamic Programming In Javascript using Tabulation, [visitado em 6 de dezembro de 2022]:

<https://www.section.io/engineering-education/dynamic-programming-in-javascript-using-tabulation/>

GEEKS (2022), What is memoization? A Complete tutorial, [visitado em 5 de dezembro de 2022]:

<https://www.geeksforgeeks.org/what-is-memoization-a-complete-tutorial/>

SKIENA (2008), Backtracking, [visitado em 3 de dezembro de 2022]:

<https://guides.codepath.com/compsci/Backtracking>

HUANG (2021), Is Recursion Really Slower than Iteration?, [visitado em 22 de novembro de 2022]:

<https://edward-huang.com/2021/02/17/is-recursion-really-slower-than-iteration/>