



Ministério da Educação  
Universidade Federal do Piauí – UFPI  
Campus Senador Helvídio Nunes de Barros – Picos  
Curso Bacharelado em Sistemas de Informação  
Professora: Juliana Oliveira de Carvalho  
Discente: Crisly Maria Silva dos Santos  
Disciplina: Estrutura de Dados II– 2023.2



## Relatório Trabalho III

### Resumo do projeto

Um grafo é uma estrutura de dados que consiste em um conjunto de nós (também chamados de vértices) e um conjunto de arestas que conectam esses nós. Cada aresta conecta dois nós e pode ser direcionada (indicando uma direção entre os nós) ou não direcionada. O projeto se concentra na exploração e compreensão de grafos e tabelas hash na linguagem de programação C. Grafos, representados por pontos e linhas, e tabelas hash, que associam chaves a valores, são estruturas de dados fundamentais em muitos campos, incluindo redes de comunicação e fluxos de informação.

O objetivo principal do estudo é verificar a eficácia dessas estruturas de dados e entender sua lógica e funcionalidade. Para isso, foram propostos três problemas específicos, que serão detalhados na seção “Seções Específicas”. Através da resolução desses problemas, pretendemos demonstrar a funcionalidade prática de grafos e tabelas hash.

Dessa forma, o trabalho realizado teve o intuito de criar resoluções de acordo com as questões propostas. A solução dos problemas propostos permitirá visualizar o funcionamento dessas estruturas de dados, proporcionando um grande aprendizado e contribuindo para a eficiência e eficácia na programação em C.

### Introdução

Neste segmento, será exposta a introdução do projeto. Grafos e tabelas de hash são estruturas de dados essenciais para o armazenamento e recuperação eficaz de dados na memória do computador. Um grafo é uma entidade que é composta por um conjunto de pontos (ou nós) e um conjunto de linhas que ligam esses pontos. Uma tabela de hash, em contrapartida, é uma espécie de estrutura que pode associar chaves a valores.

O restante do projeto visa apresentar ao leitor experimentações e explicações sobre as questões resolvidas nas estruturas de dados de grafos e tabelas de hash. A seção "Seções Específicas" irá detalhar os componentes funcionais presentes nas implementações dessas estruturas, enquanto a seção "Resultados da Execução do Programa" mostrará os resultados alcançados através da execução dos algoritmos associados a essas estruturas de dados, auxiliando o leitor

a compreender sua finalidade e eficácia. A seção "Conclusão" reiterará as informações já apresentadas no projeto, resumindo as experimentações realizadas e enfatizando a eficiência e desempenho dessas estruturas de dados. Finalmente, a seção "Apêndice" incluirá os códigos-fonte que foram desenvolvidos como parte do experimento, permitindo ao leitor uma visão mais detalhada da implementação dessas estruturas de dados.

## Seções Específicas

Neste segmento, serão apresentadas as seções específicas. Os elementos funcionais presentes em cada problema dependem do objetivo que o problema se propõe a resolver. Para este estudo, foram propostos três problemas. O primeiro visa criar um programa em C que resolva um problema muito conhecido na ciência da computação, o da Torre de Hanói. O desafio clássico da Torre de Hanói consiste em, dados  $n$  discos e 3 pinos, mover com o menor número possível de movimentos todos os  $n$  discos de um pino origem, seguindo as seguintes condições: apenas um disco pode ser movimentado de cada vez; os discos movidos deverão ser colocados sempre em discos de maior tamanho ou na base de algum pino. A Figura 1 mostra os movimentos do desafio da Torre de Hanói.

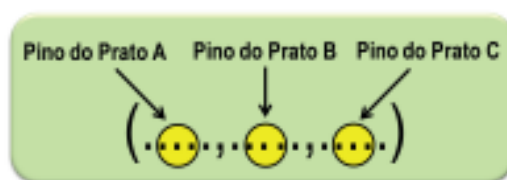


Figura 1

Por convenção uma configuração de discos será representada por um vetor com tantas posições quantas forem os discos. Na posição do disco será marcado o pino onde o disco está assentado, como mostrado na Figuras 2, 3 e 4.

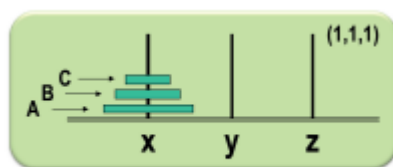


Figura 2

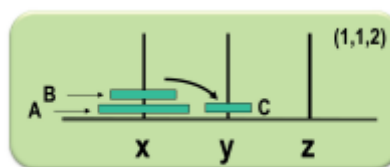


Figura 3

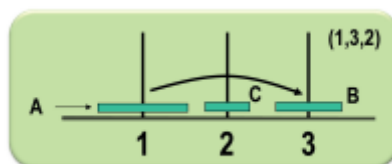


Figura 4

A Figura 5 exemplifica como formar um grafo que representa as possibilidades de movimentos dos discos. Cada vértice representa uma diferente configuração do desafio. As configurações são ligadas por arestas se uma configuração pode ser alcançada a partir de outra pelo movimento legal de um disco. De acordo com o exposto, modelar o grafo que represente o grafo de movimentos do desafio da Torre de Hanói para o caso de 4 discos. Em seguida criar o grafo usando matriz de adjacência. Depois dada uma determinada

configuração dos discos encontre o menor caminho para o resultado usando o Algoritmo de Ford-Moore-Bellman, para isso precisa-se colocar o valor 1 em cada uma das arestas. E por fim, contabilizar o tempo gasto para encontrar a solução.

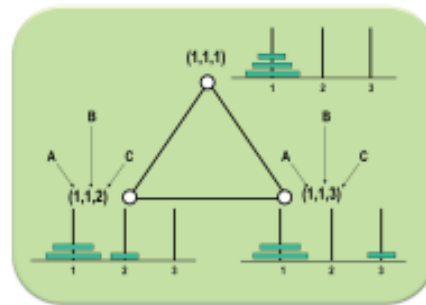


Figura 5

O segundo problema tem como objetivo implementar um algoritmo em C que suponha um grafo orientado no qual cada aresta tem um valor associado, o qual é um número real no intervalo que representa a confiabilidade de um canal de comunicação do vértice  $u$  até o vértice  $v$ . Interpretamos como a probabilidade de que o canal de  $u$  até  $v$  não venha a falhar, e supomos que essas probabilidades são independentes. Faça um programa eficiente para encontrar o caminho mais confiável entre dois vértices dados. Sabendo que a confiabilidade é dada pelo produto entre as probabilidades das arestas da origem até o destino. Quanto mais próximo de 1 a probabilidade for, mais confiável é o caminho.

O terceiro e último problema, tem a finalidade de suponha uma base de dados de 1000 funcionários e que ao invés de ordenar os dados o sistema cria uma tabela hashing para localizar os dados de um funcionário pelo número de matrícula do funcionário. Os dados dos funcionários são: Matrícula, Nome, Função, Salário. A matrícula é uma string de 6 dígitos. Faça um programa para cada um dos itens a seguir que através de uma função hashing e de uma função para colisão organize a base de dados. Depois compare os resultados através do desempenho de cada uma das soluções. E identifique qual das soluções produz um maior número de colisões. Faça o que se pede considerando um vetor destino de 101 posições e depois com 150 posições.

- (A) **Função Hashing:** rotação de 2 dígitos para a esquerda depois extrai o 2, 4 e 6 dígitos e obtenha o resto da divisão pelo tamanho do vetor destino. As colisões devem ser tratadas somando ao resto da divisão o primeiro dígito da matrícula.
- (B) **Função Hashing:** folding shift com 3 dígitos da seguinte forma: o 1, 3 e 6 ; 2, 4 e 5 , depois obtenha o resto da divisão do resultado pelo tamanho do vetor destino. As colisões devem ser realizadas somando 7 ao valor obtido.

Deixando claro na questão que em todos os casos se uma matrícula não conseguir ser colocada porque todos os possíveis locais já estão ocupados, deve-se retirar a primeira posição encontrada e colocar a nova informação.

## Resultados da Execução do Programa

Nesta seção será apresentado os resultados da execução dos programas realizados neste trabalho. A figura 1 mostra o menu com todas as funcionalidades que estão relacionadas com a primeira questão.

```
-----Menu-----  
1 - Imprimir Valores de Todos os Vertices  
2 - Imprimir um Vertice  
3 - Imprimir Matriz de Adjacencia  
4 - Imprimir Para Onde as Arestas de Vertice Estao Ligadas  
5 - Menor Caminho Utilizando Ford-Moore-Bellman  
6 - Sair  
Informe uma opcao:
```

Fig.1 - Menu

O primeiro teste a ser realizado corresponderá à opção 1 do menu, que imprimirá os valores de todos os vértices. No entanto, considerando que está sendo utilizado 81 vértices nesta questão, o print do terminal seria muito extenso para ser completamente exibido no relatório. Portanto, para manter o relatório conciso e ainda assim informativo, será apresentado uma amostra representativa para os testes subsequentes. Assim, apenas os valores de 10 vértices selecionados serão incluídos no relatório. A figura 2 apresenta isso.

```
-----Menu-----  
1 - Imprimir Valores de Todos os Vertices  
2 - Imprimir um Vertice  
3 - Imprimir Matriz de Adjacencia  
4 - Imprimir Para Onde as Arestas de Vertice Estao Ligadas  
5 - Menor Caminho Utilizando Ford-Moore-Bellman  
6 - Sair  
Informe uma opcao:  
1  
0: (1, 1, 1, 1)  
1: (1, 1, 1, 2)  
2: (1, 1, 1, 3)  
3: (1, 1, 3, 2)  
4: (1, 1, 2, 3)  
5: (1, 1, 3, 3)  
6: (1, 1, 3, 1)  
7: (1, 1, 2, 1)  
8: (1, 1, 2, 2)  
9: (1, 2, 3, 3)  
10: (1, 3, 2, 2)
```

Fig.2

Na opção 2, o usuário é solicitado a fornecer um número específico de vértice, que deve estar no intervalo de 0 a 80. Após a entrada do usuário, o programa irá imprimir os valores associados ao vértice especificado. Esses valores representam atributos ou propriedades únicas desse vértice na estrutura de dados. Observe a figura 3.

```
-----Menu-----
1 - Imprimir Valores de Todos os Vertices
2 - Imprimir um Vertice
3 - Imprimir Matriz de Adjacencia
4 - Imprimir Para Onde as Arestas de Vertice Estao Ligadas
5 - Menor Caminho Utilizando Ford-Moore-Bellman
6 - Sair
Informe uma opcao:
2
Digite o indice do vertice: 3
3: (1, 1, 3, 2)
```

Fig.3

Ao selecionar a opção 3, o programa irá gerar e exibir a matriz de adjacência correspondente a todos os vértices. No entanto, devido à limitação do terminal que não permite a visualização completa da matriz, não será viável capturar e apresentar uma imagem integral dela neste relatório.

Prosseguindo com as opções do menu, ao selecionar a opção 4, o programa irá exibir as conexões das arestas do vértice escolhido, indicando claramente para quais outros vértices ele está ligado. A Figura 4 ilustrará essa funcionalidade de maneira mais visual e compreensível.

```
-----Menu-----
1 - Imprimir Valores de Todos os Vertices
2 - Imprimir um Vertice
3 - Imprimir Matriz de Adjacencia
4 - Imprimir Para Onde as Arestas de Vertice Estao Ligadas
5 - Menor Caminho Utilizando Ford-Moore-Bellman
6 - Sair
Informe uma opcao:
4
Digite o indice do vertice: 3
1: (1, 1, 1, 2)
5: (1, 1, 3, 3)
6: (1, 1, 3, 1)
```

Fig.4

Ao selecionar a opção 5 do menu, o programa irá calcular e exibir o caminho mais curto entre um vértice de origem e um vértice de destino. Este cálculo é realizado utilizando o algoritmo Ford-Moore-Bellman. A Figura 5 demonstra um teste realizado, ilustrando tanto o caminho mais curto encontrado pelo algoritmo, quanto a distância total percorrida neste caminho e mostrando o tempo de execução em nanosegundos para encontrar o caminho.

```
1 - Imprimir Valores de Todos os Vertices
2 - Imprimir um Vertice
3 - Imprimir Matriz de Adjacencia
4 - Imprimir Para Onde as Arestas de Vertice Estao Ligadas
5 - Menor Caminho Utilizando Ford-Moore-Bellman
6 - Sair
Informe uma opcao:
5
Ford-Moore-Bellman
Digite o vertice de origem: 1
Digite o vertice de destino: 8

Caminho mais curto: 1 - 2 - 4 - 8
Distancia: 3
Tempo de execucao: 3144400.00 nanossegundos
```

Fig.5

A opção 6 do menu é destinada para encerrar o programa. Assim que o programa é finalizado, toda a memória utilizada é prontamente liberada. A Figura 6 ilustra um teste realizado, demonstrando o processo de encerramento do programa e a consequente liberação de memória.

```
-----Menu-----
1 - Imprimir Valores de Todos os Vertices
2 - Imprimir um Vertice
3 - Imprimir Matriz de Adjacencia
4 - Imprimir Para Onde as Arestas de Vertice Estao Ligadas
5 - Menor Caminho Utilizando Ford-Moore-Bellman
6 - Sair
Informe uma opcao:
6
Liberando Memoria...
Memory Liberada
```

Fig.6

A Figura 7 ilustra a estrutura de grafo empregada para solucionar a primeira questão. Os vértices e os valores atribuídos a eles neste grafo estão em conformidade com os valores especificados na questão. Esta figura serve como uma referência visual abrangente, exibindo todos os vértices e seus respectivos valores para facilitar a compreensão.

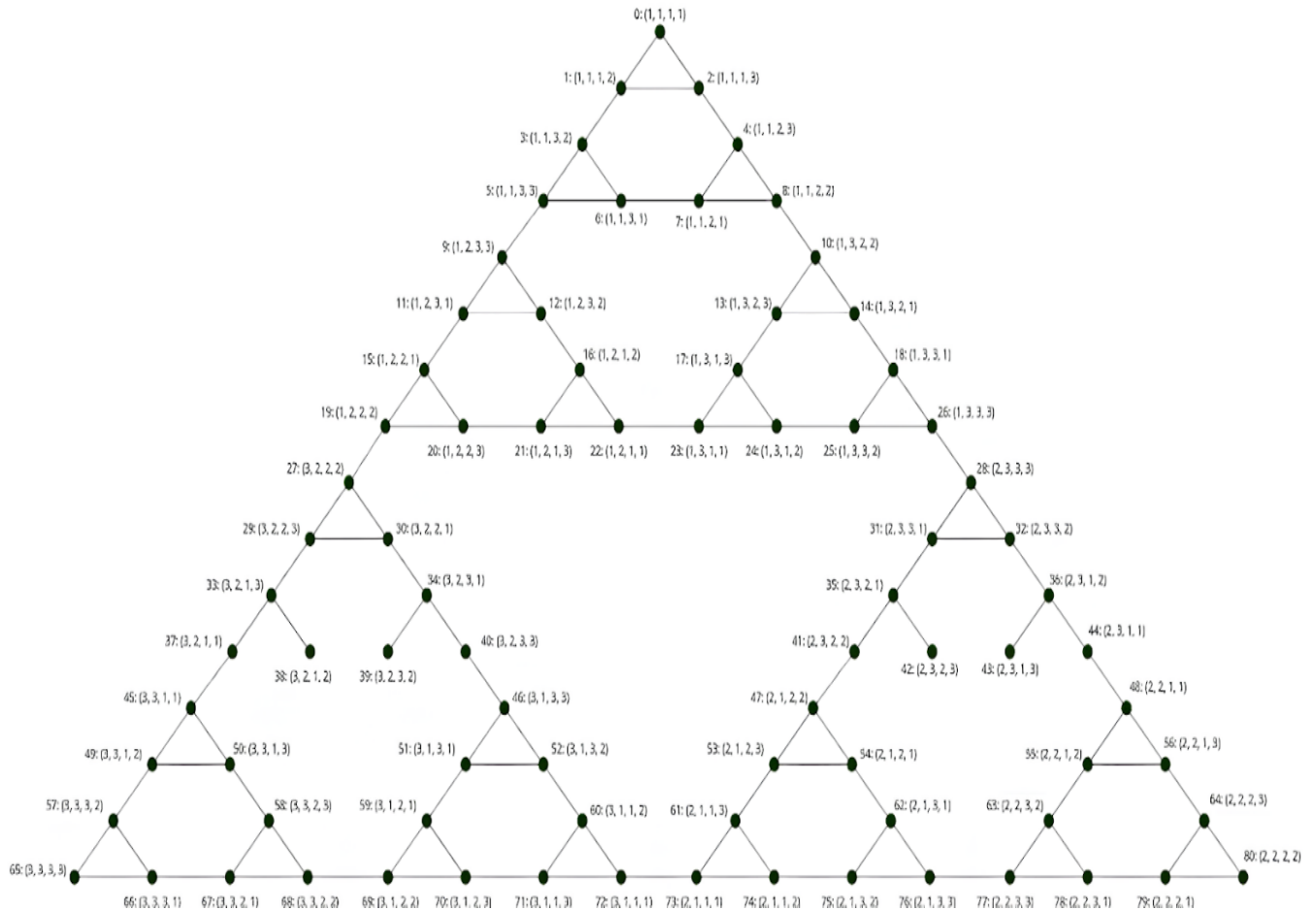


Fig.7

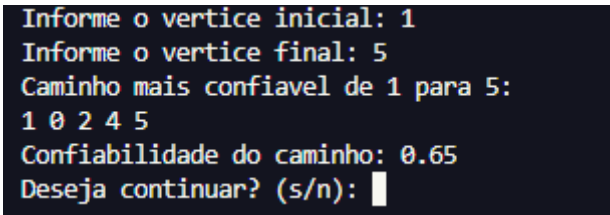
A segunda questão é focada em representar a confiabilidade de um canal de comunicação entre dois vértices. Detalhes adicionais sobre esta questão foram explicados nas “Seções Específicas”. Agora, serão apresentados os testes realizados para esta segunda questão. A Figura 8 exibirá a base de dados completa utilizada para resolver a questão. No total, foram empregados 10 vértices, cada um com seus respectivos valores, para estabelecer a comunicação de um vértice a outro.



```
1
2  adicionaAresta(grafo, 0, 1, 1.0);
3  adicionaAresta(grafo, 0, 2, 0.9);
4
5  adicionaAresta(grafo, 1, 0, 1.0);
6  adicionaAresta(grafo, 1, 2, 0.6);
7  adicionaAresta(grafo, 1, 3, 0.7);
8
9  adicionaAresta(grafo, 2, 1, 0.6);
10 adicionaAresta(grafo, 2, 0, 0.9);
11 adicionaAresta(grafo, 2, 4, 0.8);
12
13 adicionaAresta(grafo, 3, 1, 0.7);
14 adicionaAresta(grafo, 3, 4, 0.8);
15 adicionaAresta(grafo, 3, 6, 0.9);
16
17 adicionaAresta(grafo, 4, 2, 0.8);
18 adicionaAresta(grafo, 4, 3, 0.8);
19 adicionaAresta(grafo, 4, 5, 0.9);
20
21 adicionaAresta(grafo, 5, 4, 0.9);
22 adicionaAresta(grafo, 5, 6, 0.8);
23 adicionaAresta(grafo, 5, 8, 0.5);
24
25 adicionaAresta(grafo, 6, 3, 0.9);
26 adicionaAresta(grafo, 6, 5, 0.8);
27 adicionaAresta(grafo, 6, 7, 0.7);
28
29 adicionaAresta(grafo, 7, 6, 0.7);
30 adicionaAresta(grafo, 7, 8, 0.6);
31 adicionaAresta(grafo, 7, 9, 0.9);
32
33 adicionaAresta(grafo, 8, 5, 0.5);
34 adicionaAresta(grafo, 8, 7, 0.6);
35 adicionaAresta(grafo, 8, 9, 1.0);
36
37 adicionaAresta(grafo, 9, 7, 0.9);
38 adicionaAresta(grafo, 9, 8, 1.0);
```

Fig.8

O primeiro teste realizado envolveu a definição do vértice inicial como 1 e o vértice final como 5. Com base nesses parâmetros, o programa identificou o caminho mais confiável entre o vértice de origem e o de destino, e a confiabilidade dele. Este caminho está ilustrado na Figura 9 abaixo.



```
Informe o vertice inicial: 1
Informe o vertice final: 5
Caminho mais confiavel de 1 para 5:
1 0 2 4 5
Confiabilidade do caminho: 0.65
Deseja continuar? (s/n):
```



Fig.9

Um teste adicional foi conduzido, desta vez estabelecendo o vértice 0 como ponto de partida e o vértice 9 como destino. O caminho mais confiável entre esses dois pontos, bem como a confiabilidade desse caminho, será ilustrado na Figura 10.

```
Informe o vertice inicial: 0
Informe o vertice final: 9
Caminho mais confiavel de 0 para 9:
0 1 3 6 7 9
Confiabilidade do caminho: 0.40
Deseja continuar? (s/n):
```

Fig.10

O objetivo da terceira questão é inserir os números de matrículas de 1000 funcionários, realizando testes tanto em um vetor de 101 posições quanto em um de 150 posições. Em ambos os casos, as colisões que ocorrem serão devidamente tratadas. Mais detalhes dessa questão estão apresentados nas “Seções específicas” deste trabalho. Primeiramente será abordado os testes da letra A, com o vetor de 101 posições. A figura 11 mostra o Menu desta questão.

```
Menu:
1 - Cadastrar funcionarios
2 - Buscar funcionario
3 - Mostrar todos os funcionarios
4 - Sair
Escolha uma opcao:
```

Fig.11

O primeiro teste a ser realizado envolve o cadastro de um funcionário através da opção 1. Durante esse processo, serão fornecidos detalhes como o número da matrícula, nome, função e salário do funcionário. Após a inserção dessas informações, a chave do funcionário será gerada e a posição na qual foi inserida será exibida. Essa operação pode ser visualizada na Figura 12.

```
Menu:
1 - Cadastrar funcionarios
2 - Buscar funcionario
3 - Mostrar todos os funcionarios
4 - Sair
Escolha uma opcao: 1
Informe a matricula do funcionario (6 digitos): 123456
Informe o Nome: Juliana
Informe a Funcao: Professora
Informe o Salario: 10000
A chave 123456 foi inserida na posicao 18
```

Fig.12

Continuando com a opção 1, será realizado o cadastro de mais um funcionário. Este cadastro foi projetado para resultar em uma colisão, pois os cálculos foram feitos antecipadamente para garantir que isso ocorresse. Este teste é intencional para verificar se o programa está lidando corretamente com as colisões. A matrícula informada deveria ser alocada na posição 18, no entanto, como essa posição já está ocupada, a questão orienta que, em caso de colisões, deve-se somar o valor obtido com o primeiro dígito da própria matrícula. Portanto,  $18+6$  resulta na posição 24, conforme ilustrado na Figura 13.

```
Menu:
1 - Cadastrar funcionarios
2 - Buscar funcionario
3 - Mostrar todos os funcionarios
4 - Sair
Escolha uma opcao: 1
Informe a matricula do funcionario (6 digitos): 623456
Informe o Nome: Amanda
Informe a Funcao: Professora
Informe o Salario: 10000
A chave 623456 foi inserida na posicao 24
```

Fig.13

Foi realizado um teste adicional de cadastro para confirmar que os dados estão sendo inseridos na posição correta. Observe a Figura 14 abaixo para mais detalhes.

```
Menu:
1 - Cadastrar funcionarios
2 - Buscar funcionario
3 - Mostrar todos os funcionarios
4 - Sair
Escolha uma opcao: 1
Informe a matricula do funcionario (6 digitos): 723456
Informe o Nome: Eduardo
Informe a Funcao: Professor
Informe o Salario: 10000
A chave 723456 foi inserida na posicao 25
```

Fig.14

A opção 2 é destinada à busca de um funcionário. Embora essa funcionalidade não seja explicitamente solicitada na questão, ela foi implementada para fins de teste, permitindo verificar se os dados estão sendo corretamente inseridos. Conforme demonstrado na Figura 15, a inserção dos dados está ocorrendo de maneira adequada.

```
Menu:
1 - Cadastrar funcionarios
2 - Buscar funcionario
3 - Mostrar todos os funcionarios
4 - Sair
Escolha uma opcao: 2
Informe o nome do funcionario: Juliana

Funcionario Encotrado:
Matricula: 123456
Nome: Juliana
Funcao: Professora
Salario: 10000.00
Posicao: 18
```

Fig.15

A opção 3 do menu deve mostrar todos os funcionários cadastrados. Observe isso na figura 16 abaixo.

```
2 - Buscar funcionario
3 - Mostrar todos os funcionarios
4 - Sair
Escolha uma opcao: 3

Funcionarios inseridos:
Matricula: 123456
Nome: Juliana
Funcao: Professora
Salario: 10000.00
Posicao: 18
-----
Matricula: 623456
Nome: Amanda
Funcao: Professora
Salario: 10000.00
Posicao: 24
-----
Matricula: 723456
Nome: Eduardo
Funcao: Professor
Salario: 10000.00
Posicao: 25
-----
```

Fig.16

Agora, os mesmos testes serão executados, com a diferença de que as matrículas dos funcionários serão armazenadas em um vetor de 150 posições. Não há necessidade de exibir o menu, pois é idêntico ao do vetor anterior de 101 posições. O primeiro teste a ser realizado envolverá o cadastro de um funcionário. Observe a figura 17.

```
Menu:
1 - Cadastrar funcionarios
2 - Buscar funcionario
3 - Mostrar todos os funcionarios
4 - Sair
Escolha uma opcao: 1
Informe a matricula do funcionario (6 digitos): 123456
Informe o Nome: Fernando
Informe a Funcao: Diretor
Informe o Salario: 39000
A chave 123456 foi inserida na posicao 24
```

Fig.17

Agora, será realizado o cadastro de mais um funcionário, que também será alocado na posição 24. No entanto, como essa posição já está ocupada, ocorrerá uma colisão. Para resolver isso, o valor 24 será somado ao primeiro dígito da matrícula que está sendo informada. Observe a figura 18.

```
Menu:
1 - Cadastrar funcionarios
2 - Buscar funcionario
3 - Mostrar todos os funcionarios
4 - Sair
Escolha uma opcao: 1
Informe a matricula do funcionario (6 digitos): 623456
Informe o Nome: Julio
Informe a Funcao: Coordenador
Informe o Salario: 20000
A chave 623456 foi inserida na posicao 30
```

Fig.18

As funções de buscar funcionário e mostrar todos os funcionários, segue o mesmo princípio do vetor de 101 posições.

No que diz respeito às colisões, foram gerados valores aleatórios para preencher os vetores e comparar qual deles apresentou melhor desempenho. O vetor de 101 posições registrou um total de 943 colisões, um número bastante significativo. Por outro lado, o vetor de 150 posições teve 918 colisões, um valor ligeiramente menor. Com base nesses resultados, pode-se afirmar que o vetor de 150 posições apresenta um desempenho superior ao vetor de 101 posições. Observe a figura 19 e 20.

```
A chave 325968 foi inserida na posicao 21
A chave 535407 foi removida da posicao 27
A chave 735033 foi inserida na posicao 27
Numero de Colisoes: 943
```

Fig.19 – Vetor de 101 posições.

```
A chave 401017 foi removida da posicao 100
A chave 859035 foi inserida na posicao 100
A chave 898593 foi removida da posicao 95
A chave 099556 foi inserida na posicao 95
Numero de Colisoes: 918
```

Fig.20 – Vetor de 150 posições.

E por fim, a opção 4 que tem o intuito de sair do programa, e liberar a memória alocada durante a execução das funcionalidades. A figura 21 mostra isso.

```
Menu:
1 - Cadastrar funcionarios
2 - Buscar funcionario
3 - Mostrar todos os funcionarios
4 - Sair
Escolha uma opcao: 4
Liberando Memoria
```

Fig.21

Agora, será passado para os testes da parte B da terceira questão. Os detalhes específicos desses testes estão descritos na "Seção Específica" deste trabalho. O primeiro teste envolverá o cadastro de um funcionário em um vetor de 150 posições. Não há necessidade de exibir o menu, pois é o mesmo utilizado no exemplo anterior. Observe a figura 22.

```
Menu:
1 - Cadastrar funcionarios
2 - Buscar funcionario
3 - Mostrar todos os funcionarios
4 - Sair
Escolha uma opcao: 1
Informe a matricula do funcionario (6 digitos): 123456
Informe o Nome: maria
Informe a Funcao: entregadora
Informe o Salario: 1300
A chave 123456 foi inserida na posicao 81
```

Fig.22

Foi realizado um teste adicional para demonstrar o tratamento de colisões. Inicialmente, a matrícula deveria ser inserida na posição 81. No entanto, como essa posição já estava ocupada, a matrícula foi realocada para a posição 88. Conforme estipulado na questão, em caso de colisão, deve-se somar 7 à posição original. Assim, a chave, que é a matrícula, foi inserida na posição 88, conforme ilustrado na Figura 23.

```
Menu:
1 - Cadastrar funcionarios
2 - Buscar funcionario
3 - Mostrar todos os funcionarios
4 - Sair
Escolha uma opcao: 1
Informe a matricula do funcionario (6 digitos): 214365
Informe o Nome: marta
Informe a Funcao: gerente
Informe o Salario: 3000
A chave 214365 foi inserida na posicao 88
```

Fig.23

A opção 2 é destinada à busca de um funcionário. Embora essa funcionalidade não seja explicitamente solicitada na questão, ela foi implementada para fins de teste, permitindo verificar se os dados estão sendo corretamente inseridos. Conforme demonstrado na Figura 24, a inserção dos dados está ocorrendo de maneira adequada.

```
Menu:
1 - Cadastrar funcionarios
2 - Buscar funcionario
3 - Mostrar todos os funcionarios
4 - Sair
Escolha uma opcao: 2
Informe o nome do funcionario: marta

Funcionario Encotrado:
Matricula: 214365
Nome: marta
Funcao: gerente
Salario: 3000.00
Posicao: 88
```

Fig.24

A opção 3 deste programa deve mostrar todos os funcionários que foram cadastrados. Observe a figura 25 abaixo.

```

Menu:
1 - Cadastrar funcionarios
2 - Buscar funcionario
3 - Mostrar todos os funcionarios
4 - Sair
Escolha uma opcao: 3

Funcionarios inseridos:
Matricula: 123456
Nome: maria
Funcao: entregadora
Salario: 1300.00
Posicao: 81
-----
Matricula: 214365
Nome: marta
Funcao: gerente
Salario: 3000.00
Posicao: 88
-----

```

Fig.25

E por fim, a opção 4 que tem o intuito de sair do programa, e liberar a memória alocada durante a execução das funcionalidades. A figura 26 mostra isso.

```

Menu:
1 - Cadastrar funcionarios
2 - Buscar funcionario
3 - Mostrar todos os funcionarios
4 - Sair
Escolha uma opcao: 4
Liberando Memoria

```

Fig.26

Agora, prosseguiremos com os testes da parte B utilizando o vetor de 101 posições. O primeiro teste consiste em cadastrar um funcionário, fornecendo um número de matrícula de 6 dígitos. A Figura 27 ilustra a inserção do primeiro funcionário.

```

Menu:
1 - Cadastrar funcionarios
2 - Buscar funcionario
3 - Mostrar todos os funcionarios
4 - Sair
Escolha uma opcao: 1
Informe a matricula do funcionario (6 digitos): 123456
Informe o Nome: paulo
Informe a Funcao: professor
Informe o Salario: 4000
A chave 123456 foi inserida na posicao 78

```

Fig.27

Foi realizado um teste para verificar o tratamento de colisões com o primeiro cadastro. A matrícula apresentada na Figura 28 deveria, inicialmente, ser alocada na posição 78. No entanto, como essa posição já estava ocupada, a questão orienta que se deve somar 7 à posição original em caso de colisão. Portanto, a matrícula foi realocada para a posição 85, conforme demonstrado na Figura 28.

```
Menu:
1 - Cadastrar funcionarios
2 - Buscar funcionario
3 - Mostrar todos os funcionarios
4 - Sair
Escolha uma opcao: 1
Informe a matricula do funcionario (6 digitos): 214365
Informe o Nome: marcos
Informe a Funcao: gerente
Informe o Salario: 4000
A chave 214365 foi inserida na posicao 85
```

Fig.28

Outro teste realizado para verificar se está sendo inserido a chave, ou seja, a matrícula na posição correta do vetor. E como mostra ela está sendo inserida corretamente. Observe a figura 29.

```
Menu:
1 - Cadastrar funcionarios
2 - Buscar funcionario
3 - Mostrar todos os funcionarios
4 - Sair
Escolha uma opcao: 1
Informe a matricula do funcionario (6 digitos): 657431
Informe o Nome: kawan
Informe a Funcao: empresario
Informe o Salario: 5000
A chave 657431 foi inserida na posicao 2
```

Fig.29

A opção 2 é destinada à busca de um funcionário. Embora essa funcionalidade não seja explicitamente solicitada na questão, ela foi implementada para fins de teste, permitindo verificar se os dados estão sendo corretamente inseridos. Conforme demonstrado na Figura 30, a inserção dos dados está ocorrendo de maneira adequada.



```
Menu:
1 - Cadastrar funcionarios
2 - Buscar funcionario
3 - Mostrar todos os funcionarios
4 - Sair
Escolha uma opcao: 2
Informe o nome do funcionario: marcos

Funcionario Encotrado:
Matricula: 214365
Nome: marcos
Funcao: gerente
Salario: 4000.00
Posicao: 85
```

Fig.30

Seguindo com a opção 3 que é mostrar todos os funcionários cadastrados, observe que todos estão cadastrados e inseridos nas suas devidas posições. Observe a figura 31.

```
Menu:
1 - Cadastrar funcionarios
2 - Buscar funcionario
3 - Mostrar todos os funcionarios
4 - Sair
Escolha uma opcao: 3

Funcionarios inseridos:
Matricula: 657431
Nome: kawan
Funcao: empresario
Salario: 5000.00
Posicao: 2
-----
Matricula: 123456
Nome: paulo
Funcao: professor
Salario: 4000.00
Posicao: 78
-----
Matricula: 214365
Nome: marcos
Funcao: gerente
Salario: 4000.00
Posicao: 85
-----
```

Fig.31

E por fim, a opção 4 que tem o intuito de sair do programa, e liberar a memória alocada durante a execução das funcionalidades. A figura 32 mostra isso.

```
Menu:
1 - Cadastrar funcionarios
2 - Buscar funcionario
3 - Mostrar todos os funcionarios
4 - Sair
Escolha uma opcao: 4
Liberando Memoria
```

Fig.32

No que diz respeito às colisões, foram gerados valores aleatórios para preencher os vetores e comparar qual deles apresentou melhor desempenho. O vetor de 101 posições registrou um total de 943 colisões, um número bastante significativo. Por outro lado, o vetor de 150 posições teve 923 colisões, um valor ligeiramente menor. Com base nesses resultados, pode-se afirmar que o vetor de 150 posições apresenta um desempenho superior ao vetor de 101 posições. Observe a figura 33 e 34.

```
Numero de Colisoes: 923
```

Fig.33 – Vetor de 150 posições

```
Numero de Colisoes: 943
```

Fig.34 – Vetor de 101 posições

Ao comparar as funções de hashing (a) e (b) com base nos resultados obtidos nos vetores de 101 e 150 posições, observamos que ambas apresentaram uma redução no número de colisões ao aumentar o tamanho do vetor. No entanto, ao analisar a eficiência relativa entre as duas funções, é notável que a função hashing (a) teve uma diminuição de 25 colisões ao passar de um vetor de 101 para 150 posições, enquanto a função hashing (b) teve uma redução de 20 colisões na mesma transição.

Embora ambas as funções tenham demonstrado uma melhora na eficiência ao aumentar o tamanho do vetor, a função hashing (b) apresentou uma diminuição ligeiramente menor no número de colisões em comparação com a função (a). Isso sugere que, para os dados e cenários específicos analisados, a função hashing (b) teve um desempenho um pouco superior na distribuição uniforme dos valores, resultando em menos colisões.

Assim, com base nos resultados, pode-se concluir que a função hashing (b) teve um desempenho ligeiramente melhor do que a função (a) nos experimentos realizados.

## Conclusão

Este trabalho ressaltou a eficácia das implementações de estruturas de dados, como grafos e tabelas de hash, na solução de problemas. O uso dessas estruturas não só proporcionou uma organização estruturada e acessível dos dados, mas também facilitou a execução de operações de inserção e busca, reduzindo a complexidade computacional dessas operações.

Os resultados dos experimentos foram excelentes, foi possível realizar todas as questões propostas no trabalho, e com isso foi possível desenvolver programas que resolveram os problemas propostos de forma eficiente, abordando a criação e manipulação dessas estruturas de maneira precisa e efetiva. Dessa forma, a implementação de grafos e tabelas de hash apresentou desafios particulares, devido à sua complexidade e à natureza dinâmica dos dados que podem ser armazenados.

No entanto, esses desafios foram superados com êxito, resultando em soluções claras e eficientes para os problemas propostos. Em suma, a solução dos problemas permitiu não apenas o desenvolvimento de habilidades de programação relacionadas à implementação de grafos e tabelas de hash, mas também proporcionou um entendimento prático e aprofundado das estruturas de dados estudadas.

## **Apêndice**

### **Problema 1:**

```
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <time.h>
#include <windows.h>
#define INF 999999
#define TAM 81
```

```
typedef struct Vertice{
    int discos[4];
}Vertice;
```

```
typedef struct Grafo
{
    Vertice *vertices;
    int **arestas;
}Grafo;
```

```
Grafo *criargrafo(int n){
```

```
    Grafo *G;
```

```
    G = (Grafo *) malloc(sizeof(Grafo));
```

```
G->vertices = (Vertice *) malloc(n * sizeof(Vertice));
```

```
G->arestas = (int **) malloc(n * sizeof(int *));  
for(int i = 0; i < n; i++){  
    G->arestas[i] = (int *) calloc(n, sizeof(int));  
}
```

```
    return G;  
}
```

```
void ImprimirMatriz(Grafo *G, int n){  
    printf("Vertices: \n"  
    " "  
    );  
    for(int i = 0; i < n; i++){  
        printf(" %d", i);  
    }  
    printf("\n\n");  
    for(int i = 0; i < n; i++){  
        printf("%d ", i);  
        for(int j = 0; j < n; j++){  
            printf("%d ", G->arestas[i][j]);  
        }  
        printf("\n");  
    }  
    printf("\n");  
}
```

```
void InserirVerticeAresta(Grafo *G) {  
    int valores[][8] = {  
        {0, 1, 1, 1, 1, 1, 2, 9999},  
        {1, 1, 1, 1, 2, 0, 2, 3},  
        {2, 1, 1, 1, 3, 0, 1, 4},  
        {3, 1, 1, 3, 2, 1, 5, 6},  
        {4, 1, 1, 2, 3, 2, 7, 8},  
        {5, 1, 1, 3, 3, 3, 6, 9},  
        {6, 1, 1, 3, 1, 3, 5, 7},  
        {7, 1, 1, 2, 1, 4, 6, 8},  
        {8, 1, 1, 2, 2, 4, 7, 10},  
        {9, 1, 2, 3, 3, 5, 11, 12},  
    };
```

{10, 1, 3, 2, 2, 8, 13, 14},  
{11, 1, 2, 3, 1, 9, 12, 15},  
{12, 1, 2, 3, 2, 9, 11, 16},  
{13, 1, 3, 2, 3, 10, 14, 17},  
{14, 1, 3, 2, 1, 10, 13, 18},  
{15, 1, 2, 2, 1, 11, 16, 19},  
{16, 1, 2, 1, 2, 12, 21, 22},  
{17, 1, 3, 1, 3, 13, 23, 24},  
{18, 1, 3, 3, 1, 14, 25, 26},  
{19, 1, 2, 2, 2, 15, 20, 27},  
{20, 1, 2, 2, 3, 15, 19, 21},  
{21, 1, 2, 1, 3, 16, 20, 22},  
{22, 1, 2, 1, 1, 16, 21, 23},  
{23, 1, 3, 1, 1, 17, 22, 24},  
{24, 1, 3, 1, 2, 17, 23, 25},  
{25, 1, 3, 3, 2, 18, 24, 26},  
{26, 1, 3, 3, 3, 18, 25, 28},  
{27, 3, 2, 2, 2, 19, 29, 30},  
{28, 2, 3, 3, 3, 26, 31, 32},  
{29, 3, 2, 2, 3, 27, 30, 33},  
{30, 3, 2, 2, 1, 27, 29, 34},  
{31, 2, 3, 3, 1, 28, 31, 32},  
{32, 2, 3, 3, 2, 28, 31, 36},  
{33, 3, 2, 1, 3, 29, 37, 38},  
{34, 3, 2, 3, 1, 30, 39, 40},  
{35, 2, 3, 2, 1, 31, 41, 42},  
{36, 2, 3, 1, 2, 32, 43, 44},  
{37, 3, 2, 1, 1, 33, 38, 45},  
{38, 3, 2, 1, 2, 33, 37, 39},  
{39, 3, 2, 3, 2, 34, 38, 40},  
{40, 3, 2, 3, 3, 34, 39, 46},  
{41, 2, 3, 2, 2, 35, 42, 47},  
{42, 2, 3, 2, 3, 35, 41, 43},  
{43, 2, 3, 1, 3, 36, 42, 44},  
{44, 2, 3, 1, 1, 36, 43, 48},  
{45, 3, 3, 1, 1, 37, 49, 50},  
{46, 3, 1, 3, 3, 40, 51, 52},  
{47, 2, 1, 2, 2, 41, 53, 54},  
{48, 2, 2, 1, 1, 44, 55, 56},  
{49, 3, 3, 1, 2, 45, 50, 57},  
{50, 3, 3, 1, 3, 45, 49, 58},  
{51, 3, 1, 3, 1, 46, 52, 59},  
{52, 3, 1, 3, 2, 46, 51, 60},

```

    {53, 2, 1, 2, 3, 47, 54, 61},
    {54, 2, 1, 2, 1, 47, 53, 62},
    {55, 2, 2, 1, 2, 48, 56, 63},
    {56, 2, 2, 1, 3, 48, 55, 64},
    {57, 3, 3, 3, 2, 49, 65, 66},
    {58, 3, 3, 2, 3, 50, 67, 68},
    {59, 3, 1, 2, 1, 51, 69, 70},
    {60, 3, 1, 1, 2, 52, 71, 72},
    {61, 2, 1, 1, 3, 53, 73, 74},
    {62, 2, 1, 3, 1, 54, 75, 76},
    {63, 2, 2, 3, 2, 55, 77, 78},
    {64, 2, 2, 2, 3, 56, 79, 80},
    {65, 3, 3, 3, 3, 57, 66, 9999},
    {66, 3, 3, 3, 1, 57, 65, 67},
    {67, 3, 3, 2, 1, 58, 66, 68},
    {68, 3, 3, 2, 2, 58, 67, 69},
    {69, 3, 1, 2, 2, 59, 68, 70},
    {70, 3, 1, 2, 3, 59, 69, 71},
    {71, 3, 1, 1, 3, 60, 70, 72},
    {72, 3, 1, 1, 1, 60, 71, 73},
    {73, 2, 1, 1, 1, 61, 72, 74},
    {74, 2, 1, 1, 2, 61, 73, 75},
    {75, 2, 1, 3, 2, 62, 74, 76},
    {76, 2, 1, 3, 3, 62, 75, 77},
    {77, 2, 2, 3, 3, 63, 76, 78},
    {78, 2, 2, 3, 1, 63, 77, 79},
    {79, 2, 2, 2, 1, 64, 78, 80},
    {80, 2, 2, 2, 2, 64, 79, 9999}
};

```

```
int tamanhoValores = 81;
```

```
int i = 0;
```

```

while (i < tamanhoValores) {
    int verticeAtual = valores[i][0];
    int disco1 = valores[i][1];
    int disco2 = valores[i][2];
    int disco3 = valores[i][3];
    int disco4 = valores[i][4];
    int aresta1 = valores[i][5];
    int aresta2 = valores[i][6];
    int aresta3 = valores[i][7];
}

```

```

G->vertices[verticeAtual].discos[0] = disco1;
G->vertices[verticeAtual].discos[1] = disco2;
G->vertices[verticeAtual].discos[2] = disco3;
G->vertices[verticeAtual].discos[3] = disco4;

G->arestas[verticeAtual][aresta1] = 1;
G->arestas[verticeAtual][aresta2] = 1;
if (aresta3 != 9999)
    G->arestas[verticeAtual][aresta3] = 1;

    i++;
}

}

void ImprimirVertice(Grafo *G, int linha){
    printf("%d: (%d, %d, %d, %d)\n", linha, G->vertices[linha].discos[0], G-
>vertices[linha].discos[1], G->vertices[linha].discos[2], G-
>vertices[linha].discos[3]);
}

void Menorcaminho_DijsktranBelman(int vertices, int destino, int
vertice_anterior[], int distancia[]){

    printf("\nCaminho mais curto: ");
    int comprimentocaminho = 0;
    int *caminhovertices; caminhovertices = (int*)malloc(vertices * sizeof(int));

    int verticeatual = destino;

    while (verticeatual != -1) {
        caminhovertices[comprimentocaminho++] = verticeatual;
        verticeatual = vertice_anterior[verticeatual];
    }

    for (int i = comprimentocaminho - 1; i >= 0; i--) {
        printf("%d ", caminhovertices[i]);
        if (i > 0)
            printf(" - ");
    }
    printf("\nDistancia: %d\n", distancia[destino]);
    free(caminhovertices);
}

```

```
}
```

```
void bellmanFord(int **grafo, int vertices, int origem, int destino) {
    int temciclonegativo = 0;
    int *distancia, *vertice_anterior;
    distancia = (int*)malloc(vertices * sizeof(int));
    vertice_anterior = (int*)malloc(vertices * sizeof(int));

    for (int i = 0; i < vertices; i++) {
        distancia[i] = INF;
        vertice_anterior[i] = -1;
    }

    distancia[origem] = 0;

    for (int qtdRelaxamentos = 0; qtdRelaxamentos < vertices - 1;
    qtdRelaxamentos++) {
        for (int i = 0; i < vertices; i++) {
            for (int j = 0; j < vertices; j++) {
                if (grafo[i][j] && distancia[i] != INF && distancia[i] + grafo[i][j] <
                distancia[j]) {
                    distancia[j] = distancia[i] + grafo[i][j];
                    vertice_anterior[j] = i;
                }
            }
        }
    }

    for (int i = 0; i < vertices; i++) {
        for (int j = 0; j < vertices; j++) {
            if (grafo[i][j] && distancia[i] != INF && distancia[i] + grafo[i][j] <
            distancia[j]) {
                printf("O grafo contem um ciclo negativo.\n");
                temciclonegativo = 1;
                j = vertices;
                i = j;
            }
        }
    }

    if(!temciclonegativo)
```



```

Menorcaminho_DijkstraBelman(vertices,destino,vertice_anterior,distancia);
    free(distancia);
    free(vertice_anterior);
}

```

```

void liberarMemoria(Grafo **G, int qtdVertice){
    for (int i = 0; i < qtdVertice; i++){
        free((*G)->arestas[i]);
        (*G)->arestas[i] = NULL;
    }
    free((*G)->vertices);
    (*G)->vertices = NULL;

    free(*G);
    *G = NULL;
}

```

```

void lugarDasArestas(int *vetorVertice, int *resposta, int *cont){
    *cont = 0;

    for(int i = 0; i < 81; i++){
        if(vetorVertice[i] == 1){
            if(*cont == 0)
                resposta[0] = i;
            if(*cont == 1)
                resposta[1] = i;
            if(*cont == 2)
                resposta[2] = i;
            *cont += 1;
        }
    }
}

```

```

int verificarOpcaoPraOndeIr(int *resposta, int vertice){
    int cont = 1;
    if (vertice == resposta[0])
        cont = 0;
    if (vertice == resposta[1])
        cont = 0;
    if (vertice == resposta[2])

```

```

        cont = 0;

    return cont;
}

int main()
{
    Grafo *G;
    int op = 0, vertice, cont, resposta[3], origem, destino;

    G = criargrafo(81);

    InsereVerticeAresta(G);

    while (op != 6)
    {
        printf("1 - Imprimir Valores de Todos os Vertices\n");
        printf("2 - Imprimir um Vertice\n");
        printf("3 - Imprimir Matriz de Adjacencia\n");
        printf("4 - Imprimir Para Onde as Arestas de Vertice Estao Ligadas\n");
        printf("5 - Menor Caminho Utilizando Ford-Moore-Bellman\n");
        printf("6 - Sair\n");
        printf("Informe uma opcao:\n");
        scanf("%d", &op);

        switch (op)
        {
            case 1:{
                for (int i = 0; i < TAM; i++)
                    ImprimirVertice(G, i);
                break;
            }
            case 2:{
                printf("Digite o indice do vertice: ");
                scanf("%d", &vertice);
                if (vertice >= 0 && vertice <= 80)
                    ImprimirVertice(G, vertice);
                else

```

```

        printf("Indice de vertice não existe.\n");
        break;
    }
    case 3:{
        ImprimirMatriz(G, TAM);
        break;
    }
    case 4:{
        cont = 0;
        printf("Digite o indice do vertice: ");
        scanf("%d", &vertice);

        if (vertice >= 0 && vertice <= 80){
            lugarDasArestas(G->arestas[vertice], resposta, &cont);

            ImprimirVertice(G, resposta[0]);
            ImprimirVertice(G, resposta[1]);

            if (cont == 3)
            {
                ImprimirVertice(G, resposta[2]);
            }
            memset(resposta, 0, sizeof(resposta));
        }
        else
            printf("Indice de vertice não existe.\n");
        break;
    }

    case 5:{
        double elapsed_nanos;
        LARGE_INTEGER start, end, frequency;

        printf("Ford-Moore-Bellman\n");
        printf("Digite o vertice de origem: ");
        scanf("%d", &origem);
        printf("Digite o vertice de destino: ");
        scanf("%d", &destino);

        if ((origem >= 0 && origem <= 80) && (destino >= 0 && destino <=
80)){
            QueryPerformanceFrequency(&frequency);

```

```

        QueryPerformanceCounter(&start);
        bellmanFord(G->arestas, TAM, origem, destino);
        QueryPerformanceCounter(&end);

        elapsed_nanos = ((end.QuadPart - start.QuadPart) * 1.0 /
frequency.QuadPart) * 1000000000;
        printf("Tempo de execucao: %.2f nanossegundos\n",
elapsed_nanos);
    }
    else{
        printf("Verifique se os vertices informados existem\n");
    }
    break;
}

case 6:{
    break;
}

default:
    printf("Opcao Invalida\n");
    break;
}
}
liberarMemoria(&G, 81);
return 0;
}

```

## Problema 2:

```

#include <stdio.h>
#include <stdlib.h>

#define INF 1e9

typedef struct{
    int v;
    double w;
} Aresta;

typedef struct no{
    Aresta aresta;
    struct no *proximo;
}

```

```
} No;
```

```
typedef struct{  
    No **cabeca;  
    int n;  
} Grafo;
```

```
Grafo *criaGrafo(int n){  
    Grafo *grafo = (Grafo *)malloc(sizeof(Grafo));  
    grafo->cabeca = (No **)malloc(n * sizeof(No *));  
    grafo->n = n;  
    for (int i = 0; i < n; i++){  
        grafo->cabeca[i] = NULL;  
    }  
    return grafo;  
}
```

```
void adicionaAresta(Grafo *grafo, int u, int v, double w){  
    No *novoNo = (No *)malloc(sizeof(No));  
    novoNo->aresta.v = v;  
    novoNo->aresta.w = w;  
    novoNo->proximo = grafo->cabeca[u];  
    grafo->cabeca[u] = novoNo;  
}
```

```
void imprimeCaminhoRecursivamente(int dest, int pred[]){  
    if (dest == -1){  
        return;  
    }  
    imprimeCaminhoRecursivamente(pred[dest], pred);  
    printf("%d ", dest);  
}
```

```
void dijkstra(Grafo *grafo, int src, int dest){  
    double dist[grafo->n];  
    int pred[grafo->n], visitado[grafo->n];  
  
    for (int i = 0; i < grafo->n; i++){  
        dist[i] = 0;  
        pred[i] = -1;  
        visitado[i] = 0;  
    }
```

```

dist[src] = 1;
for (int i = 0; i < grafo->n; i++){
    int u = -1;
    for (int j = 0; j < grafo->n; j++)
        if (!visitado[j] && (u == -1 || dist[j] > dist[u]))
            u = j;
    if (dist[u] == 0){
        break;
    }
    visitado[u] = 1;

```

```

    No *temp = grafo->cabeca[u];
    while (temp){
        int v = temp->aresta.v;
        double w = temp->aresta.w;
        if (!visitado[v] && dist[v] < dist[u] * w){
            dist[v] = dist[u] * w;
            pred[v] = u;
        }
        temp = temp->proximo;
    }
}

```

```

printf("Caminho mais confiavel de %d para %d:\n", src, dest);
imprimeCaminhoRecursivamente(dest, pred);
printf("\n");

```

```

    printf("Confiabilidade do caminho: %.2f\n", dist[dest]);
}

```

```

void liberaMemoria(Grafo *grafo){
    for (int i = 0; i < grafo->n; i++)
    {
        No *no = grafo->cabeca[i];
        while (no)
        {
            No *temp = no;
            no = no->proximo;
            free(temp);
        }
    }
    free(grafo->cabeca);
}

```

```
    free(grafo);  
}
```

```
int main(){  
  
    int n = 12;  
    Grafo *grafo = criaGrafo(n);  
  
    adicionaAresta(grafo, 0, 1, 1.0);  
    adicionaAresta(grafo, 0, 2, 0.9);  
  
    adicionaAresta(grafo, 1, 0, 1.0);  
    adicionaAresta(grafo, 1, 2, 0.6);  
    adicionaAresta(grafo, 1, 3, 0.7);  
  
    adicionaAresta(grafo, 2, 1, 0.6);  
    adicionaAresta(grafo, 2, 0, 0.9);  
    adicionaAresta(grafo, 2, 4, 0.8);  
  
    adicionaAresta(grafo, 3, 1, 0.7);  
    adicionaAresta(grafo, 3, 4, 0.8);  
    adicionaAresta(grafo, 3, 6, 0.9);  
  
    adicionaAresta(grafo, 4, 2, 0.8);  
    adicionaAresta(grafo, 4, 3, 0.8);  
    adicionaAresta(grafo, 4, 5, 0.9);  
  
    adicionaAresta(grafo, 5, 4, 0.9);  
    adicionaAresta(grafo, 5, 6, 0.8);  
    adicionaAresta(grafo, 5, 8, 0.5);  
  
    adicionaAresta(grafo, 6, 3, 0.9);  
    adicionaAresta(grafo, 6, 5, 0.8);  
    adicionaAresta(grafo, 6, 7, 0.7);  
  
    adicionaAresta(grafo, 7, 6, 0.7);  
    adicionaAresta(grafo, 7, 8, 0.6);  
    adicionaAresta(grafo, 7, 9, 0.9);  
}
```

```

adicionaAresta(grafo, 8, 5, 0.5);
adicionaAresta(grafo, 8, 7, 0.6);
adicionaAresta(grafo, 8, 9, 1.0);

adicionaAresta(grafo, 9, 7, 0.9);
adicionaAresta(grafo, 9, 8, 1.0);

do
{
    int Vinicial, Vfinal;
    printf("Informe o vertice inicial: ");
    scanf("%d", &Vinicial);
    printf("Informe o vertice final: ");
    scanf("%d", &Vfinal);
    dijkstra(grafo, Vinicial, Vfinal);
    printf("Deseja continuar? (s/n): ");
    getchar();

} while (getchar() == 's' || getchar() == 'S');

liberaMemoria(grafo);

return 0;
}

```

### Problema 3 letra A:

```

#include <stdio.h>
#include <stdlib.h>
#include <string.h>

#define TAM 150 //150
int quantFunc = 1000;
typedef struct
{
    char matricula[7];
    char nome[50];
    char funcao[50];
}

```



```
    float salario;  
} Funcionario;
```

```
typedef struct  
{  
    Funcionario *tabela[TAM];  
    int colisoos;  
} TabelaHash;
```

```
int hash(char matricula[7])  
{  
    char chave[7];  
    sprintf(chave, "%c%c%c", matricula[4], matricula[5], matricula[6]);  
    int d2 = chave[1] - '0';  
    int d4 = chave[3] - '0';  
    int d6 = chave[5] - '0';  
    int valor_hash = d2 * 100 + d4 * 10 + d6;  
    int posicao = valor_hash % TAM;  
    return posicao;  
}
```

```
void inserir_funcionario(TabelaHash *tabela, char *matricula, char *nome, char  
*funcao, float salario)
```

```
{  
    int posicao = hash(matricula);  
    int primeiro_digito = matricula[0] - '0';
```

```
    int posicao_remove = -1;
```

```
    if (tabela->tabela[posicao] != NULL)  
    {
```

```
        tabela->colisoos++;  
        int posicao_inicial = posicao;
```

```
        while (tabela->tabela[posicao] != NULL){  
            if (posicao_remove == -1 && tabela->tabela[posicao] != NULL){  
                posicao_remove = posicao;  
            }  
        }
```

```
        posicao = (posicao + primeiro_digito) % TAM;
```

```

        if (posicao == posicao_inicial){

            if (tabela->tabela[posicao_remove] != NULL){
                printf("A chave %s foi removida da posicao %d\n", tabela-
>tabela[posicao_remove]->matricula, posicao_remove);
                free(tabela->tabela[posicao_remove]);
                tabela->tabela[posicao_remove] = NULL;
            }
            break;
        }
    }
}

tabela->tabela[posicao] = malloc(sizeof(Funcionario));

strcpy(tabela->tabela[posicao]->matricula, matricula);
strcpy(tabela->tabela[posicao]->nome, nome);
strcpy(tabela->tabela[posicao]->funcao, funcao);
tabela->tabela[posicao]->salario = salario;

printf("A chave %s foi inserida na posicao %d\n", matricula, posicao);
}

void mostrar_funcionarios_inseridos(TabelaHash *tabela)
{
    printf("Funcionarios inseridos:\n");
    for (int i = 0; i < TAM; i++)
    {
        if (tabela->tabela[i] != NULL)
        {
            printf("Matricula: %s\n", tabela->tabela[i]->matricula);
            printf("Nome: %s\n", tabela->tabela[i]->nome);
            printf("Funcao: %s\n", tabela->tabela[i]->funcao);
            printf("Salario: %.2f\n", tabela->tabela[i]->salario);
            printf("Posicao: %d\n", i);
            printf("-----\n");
        }
    }
}

```

//FUNCAO TESTE PARA SABER SE ESTAVA INSERINDO  
CORRETAMENTE

```
void buscar_funcionario(TabelaHash *tabela, char *nome)
{
    int encontrado = 0;

    for (int i = 0; i < TAM; i++)
    {
        if (tabela->tabela[i] != NULL && strcmp(tabela->tabela[i]->nome, nome)
== 0)
        {
            printf("Funcionario Encotrado:\n");
            printf("Matricula: %s\n", tabela->tabela[i]->matricula);
            printf("Nome: %s\n", tabela->tabela[i]->nome);
            printf("Funcao: %s\n", tabela->tabela[i]->funcao);
            printf("Salario: %.2f\n", tabela->tabela[i]->salario);
            printf("Posicao: %d\n", i);

            encontrado = 1;
            break;
        }
    }

    if (encontrado == 0){
        printf("Funcionario com nome %s nao encontrado.\n", nome);
    }
}
```

```
void liberar_memoria(TabelaHash *tabela)
{
    for (int i = 0; i < TAM; i++)
    {
        if (tabela->tabela[i] != NULL)
        {
            free(tabela->tabela[i]);
        }
    }
}
```

```

int main()
{
    TabelaHash tabela;
    int i;

    tabela.colisoos = 0;
    for (i = 0; i < TAM; i++)
        tabela.tabela[i] = NULL;

    char continuar_programa = 's';

    while (continuar_programa == 's' || continuar_programa == 'S')
    {
        int opcao;

        printf("\nMenu:\n");
        printf("1 - Cadastrar funcionarios\n");
        printf("2 - Buscar funcionario\n");
        printf("3 - Mostrar todos os funcionarios\n");
        printf("4 - Sair\n");
        printf("Escolha uma opcao: ");
        scanf("%d", &opcao);

        switch (opcao)
        {
            case 1:{
                if(quantFunc == 0){
                    printf("Atingindo o Limite de Cadastramento dos Funcionarios\n");
                }
                else{
                    char matricula[7];
                    char nome[50];
                    char funcao[50];
                    float salario;

                    do {
                        printf("Informe a matricula do funcionario (6 digitos): ");
                        scanf("%s", matricula);
                        getchar();
                    } while (strlen(matricula) != 6);

                    printf("Informe o Nome: ");

```

```

    fflush(stdin);
    scanf("%s", nome);

    printf("Informe a Funcao: ");
    fflush(stdin);
    scanf("%s", funcao);

    printf("Informe o Salario: ");
    scanf("%f", &salario);

    inserir_funcionario(&tabela, matricula, nome, funcao, salario);
    quantFunc -= 1;
}
break;
}

case 2:{
    // Implementar a busca do funcionário
    char nome[50];
    printf("Informe o nome do funcionario: ");
    fflush(stdin);
    scanf("%s", nome);
    printf("\n");
    buscar_funcionario(&tabela, nome);
    break;
}
case 3:
    printf("\n");
    mostrar_funcionarios_inseridos(&tabela);
    break;
case 4:
    continuar_programa = 'n';
    break;
default:
    printf("Opcao invalida. Tente novamente.\n");
    break;
}
}
printf("Liberando Memoria\n");
liberar_memoria(&tabela);

```

```
    return 0;
}
```

### **Problema 03 LETRA B:**

```
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
```

```
#define TAM 150 //101
int quantFunc = 1000;
```

```
typedef struct
{
    char matricula[7];
    char nome[50];
    char funcao[50];
    float salario;
} Funcionario;
```

```
typedef struct
{
    Funcionario *tabela[TAM];
    int colisoos;
} TabelaHash;
```

```
int hash(char *matricula)
{
    int digito1 = matricula[0] - '0';
    int digito3 = matricula[2] - '0';
    int digito6 = matricula[5] - '0';
```

```
    int digito2 = matricula[1] - '0';
    int digito4 = matricula[3] - '0';
    int digito5 = matricula[4] - '0';
```

```
    int soma = digito1 * 100 + digito3 * 10 + digito6 + digito2 * 100 + digito4 *
10 + digito5;
```

```

    int posicao = soma % TAM;

    return posicao;
}
void inserir_funcionario(TabelaHash *tabela, char *matricula, char *nome, char
*funcao, float salario)
{
    int posicao = hash(matricula);

    int posicao_remove = -1;

    if (tabela->tabela[posicao] != NULL){
        tabela->colisoes++;
        int posicao_inicial = posicao;

        while (tabela->tabela[posicao] != NULL){
            if (posicao_remove == -1 && tabela->tabela[posicao] != NULL){
                posicao_remove = posicao;
            }

            posicao = (posicao + 7) % TAM;

            if (posicao == posicao_inicial){

                if (tabela->tabela[posicao_remove] != NULL){
                    printf("A chave %s foi removida da posicao %d\n", tabela-
>tabela[posicao_remove]->matricula, posicao_remove);
                    free(tabela->tabela[posicao_remove]);
                    tabela->tabela[posicao_remove] = NULL;
                }
                break;
            }
        }
    }

    tabela->tabela[posicao] = malloc(sizeof(Funcionario));

```

```

strcpy(tabela->tabela[posicao]->matricula, matricula);
strcpy(tabela->tabela[posicao]->nome, nome);
strcpy(tabela->tabela[posicao]->funcao, funcao);
tabela->tabela[posicao]->salario = salario;

printf("A chave %s foi inserida na posicao %d\n", matricula, posicao);
}

```

```

void mostrar_funcionarios_inseridos(TabelaHash *tabela)
{
    printf("\nFuncionarios inseridos:\n");
    for (int i = 0; i < TAM; i++){
        if (tabela->tabela[i] != NULL){
            printf("Matricula: %s\n", tabela->tabela[i]->matricula);
            printf("Nome: %s\n", tabela->tabela[i]->nome);
            printf("Funcao: %s\n", tabela->tabela[i]->funcao);
            printf("Salario: %.2f\n", tabela->tabela[i]->salario);
            printf("Posicao: %d\n", i);
            printf("-----\n");
        }
    }
}

```

//FUNCAO TESTE PARA SABER SE ESTAVA INSERINDO CORRETAMENTE

```

void buscar_funcionario(TabelaHash *tabela, char *nome)
{
    int encontrado = 0;

    for (int i = 0; i < TAM; i++)
    {
        if (tabela->tabela[i] != NULL && strcmp(tabela->tabela[i]->nome, nome)
== 0)
        {
            printf("Funcionario Encotrado:\n");
            printf("Matricula: %s\n", tabela->tabela[i]->matricula);
            printf("Nome: %s\n", tabela->tabela[i]->nome);
            printf("Funcao: %s\n", tabela->tabela[i]->funcao);

```



```

        printf("Salario: %.2f\n", tabela->tabela[i]->salario);
        printf("Posicao: %d\n", i);

        encontrado = 1;
        break;
    }
}

if (encontrado == 0){
    printf("Funcionario com nome %s nao encontrado.\n", nome);
}
}

```

```

void liberar_memoria(TabelaHash *tabela)
{
    for (int i = 0; i < TAM; i++)
    {
        if (tabela->tabela[i] != NULL)
        {
            free(tabela->tabela[i]);
        }
    }
}

```

```

int main()
{
    TabelaHash tabela;
    int i;

    tabela.colisoes = 0;
    for (i = 0; i < TAM; i++)
        tabela.tabela[i] = NULL;

    char continuar_programa = 's';

    while (continuar_programa == 's' || continuar_programa == 'S')

```

```

{
    int opcao;

    printf("\nMenu:\n");
    printf("1 - Cadastrar funcionarios\n");
    printf("2 - Buscar funcionario\n");
    printf("3 - Mostrar todos os funcionarios\n");
    printf("4 - Sair\n");
    printf("Escolha uma opcao: ");
    scanf("%d", &opcao);

    switch (opcao)
    {
    case 1:{
        if(quantFunc == 0){
            printf("Atingido o Limite de Cadastramento dos Funcionarios\n");
        }
        else{
            char matricula[7];
            char nome[50];
            char funcao[50];
            float salario;

            do {
                printf("Informe a matricula do funcionario (6 digitos): ");
                scanf("%s", matricula);
                getchar();
            } while (strlen(matricula) != 6);

            printf("Informe o Nome: ");
            fflush(stdin);
            scanf("%[^\\n]", nome);

            printf("Informe a Funcao: ");
            fflush(stdin);
            scanf("%[^\\n]", funcao);

            printf("Informe o Salario: ");
            scanf("%f", &salario);

            inserir_funcionario(&tabela, matricula, nome, funcao, salario);

```

```

        quantFunc -=1;

    }
    break;
}

case 2:{

    char nome[50];
    printf("Informe o nome do funcionario: ");
    fflush(stdin);
    scanf("%[^\\n]", nome);
    printf("\\n");
    buscar_funcionario(&tabela, nome);
    break;
}
case 3:
    mostrar_funcionarios_inseridos(&tabela);
    break;
case 4:
    continuar_programa = 'n';
    break;
default:
    printf("Opcao invalida. Tente novamente.\\n");
    break;
}
}

```

```

printf("Liberando Memoria\\n");
liberar_memoria(&tabela);

```

```

return 0;

```

```

}

```

