UNIVERSIDADE FEDERAL RURAL DO SEMI-ÁRIDO - UFERSA

DEPARTAMENTO DE ENGENHARIAS E TECNOLOGIA

DICIONÁRIO COM TABELA *HASH*

Pau dos Ferros – RN

Julho 2025

IZADORA LOUYZA SILVA FIGUEIREDO
GitMaster

LÍVIAN MARIA LUCENA GOMES PINHEIRO
Redatora

VICTOR HUGO DE OLIVEIRA
Codificador e apresentador

DICIONÁRIO COM TABELA *HASH*

Trabalho apresentado à disciplina Laboratório de Algoritmos e Estrutura de Dados II, ministrada pelo Prof. Dr. Kennedy Reurison Lopes, do curso de Tecnologia da Informação, como requisito parcial para a composição da nota referente à segunda e terceira unidades.

Pau dos Ferros – RN
Julho 2025

Sumário

1 Introdução

As estruturas de dados desempenham papel essencial no desenvolvimento de algoritmos eficientes, influenciando diretamente o desempenho em operações de consulta e manipulação de dados. Entre essas estruturas, destaca-se a tabela *hash*, uma estrutura de dispersão que permite acessos rápidos, além de otimizar o uso da memória.

Neste contexto, o presente trabalho apresenta a implementação de um dicionário digital desenvolvido em linguagem C, que utiliza a tabela *hash* para gerenciar palavras e seus respectivos significados. A aplicação desenvolvida possibilita ao usuário cadastrar novos termos, consultar seus significados, remover palavras, exibir o conteúdo armazenado e salvar os dados em um arquivo de texto, garantindo a persistência das informações.

A escolha pela tabela *hash* fundamenta-se em sua simplicidade e comprovado desempenho. Segundo (BENTO; PEREIRA DE SÁ; SZWARCFITER, 2012), essa estrutura é amplamente utilizada em sistemas que demandam associação e busca eficientes, possibilitando operações rápidas sem grande custo computacional. Neste projeto, a função de *hash* converte as palavras em índices numéricos para armazenamento direto, enquanto colisões são tratadas por sondagem linear, assegurando a integridade dos dados.

Dessa forma, este trabalho tem como objetivo implementar um dicionário digital utilizando tabela *hash*, focando na construção e funcionamento dessa estrutura para armazenamento e consulta de palavras e seus significados.

2 Descrição do Problema

Em aplicações que lidam com grandes volumes de dados textuais, a eficiência nas operações de busca, inserção e remoção é fundamental para garantir o bom desempenho do sistema. Estruturas de dados simples, como listas encadeadas ou *arrays*, apresentam limitações nessas operações, pois tendem a exigir buscas lineares que aumentam o tempo de processamento conforme o volume de dados cresce.

Além disso, a necessidade de manter os dados organizados para acesso rápido é fundamental em contextos onde o usuário espera respostas imediatas, como em consultas interativas. A demora nas operações compromete a experiência do usuário e pode tornar inviável a utilização do sistema em larga escala.

Para contornar essas limitações, é essencial utilizar estruturas que proporcionem acesso direto ou próximo disso, diminuindo o tempo gasto em buscas e atualizações de dados. Como reforçam (BENTO; PEREIRA DE SÁ; SZWARCFITER, 2012), a técnica de *hashing* se destaca nesse cenário, pois mapeia os dados de entrada (neste caso, palavras) para índices numéricos em

uma tabela de tamanho fixo, permitindo acessos em tempo, em geral, constante $O(1)$.

Entretanto, o uso de tabelas *hash* traz desafios dentro da aplicação, como o tratamento de colisões, que ocorrem quando diferentes palavras resultam no mesmo índice. Métodos eficientes para resolver essas colisões, como a sondagem linear, são fundamentais para manter o desempenho esperado e garantir a integridade das informações armazenadas. A forma como essas colisões são tratadas será explorada e justificada ao longo do desenvolvimento deste trabalho.

Dessa forma, o problema a ser resolvido consiste em desenvolver um dicionário digital baseado em tabela *hash* que:

- Permita a inserção de palavras e seus respectivos significados;
- Ofereça consultas rápidas aos termos cadastrados;
- Possibilite a remoção de palavras sem comprometer a estrutura da tabela;
- Permita a visualização de todas as palavras e seus significados cadastrados;
- Garanta a persistência dos dados por meio de armazenamento em um arquivo de texto.

3 Solução Proposta

O sistema foi desenvolvido em linguagem C, utilizando uma tabela *hash* de tamanho fixo com 19 posições para armazenar palavras e seus respectivos significados. Cada posição da tabela guarda uma palavra e seu significado por meio de uma estrutura do tipo *struct*. O programa permite ao usuário, por meio de funções, cadastrar novas palavras, buscar significados, remover entradas, exibir todas as palavras cadastradas e salvar os dados em um arquivo de texto.

A função *hash* implementada consiste na soma dos valores numéricos (códigos ASCII) de cada caractere da palavra. O total é, então, dividido pelo tamanho da tabela (19), e o resto da divisão define o índice onde a palavra será armazenada. Por exemplo, palavras que são anagramas, como ”amor” e ”roma”, embora tenham significados diferentes, possuem os mesmos caracteres em ordens diferentes, resultando na mesma soma ASCII e, consequentemente, no mesmo índice da tabela, o que causa uma colisão.

Esse tipo de mapeamento pode ocasionar colisões, que ocorrem quando diferentes palavras são atribuídas à mesma posição na tabela. Para tratar essas colisões, foi adotada a técnica de sondagem linear, na qual o programa verifica a próxima posição disponível sempre que a posição calculada já estiver ocupada, repetindo esse processo até encontrar um espaço livre.

A principal vantagem da sondagem linear é a ausência de referências externas, o que permite que a tabela *hash* utilize uma quantidade maior de posições dentro da mesma quantidade de

memória disponível. Essa simplicidade de implementação torna o método eficiente para tabelas de tamanho fixo. Contudo, é importante observar que a tabela pode ficar totalmente cheia, impossibilitando novas inserções, e que o método tende a formar agrupamento primário, ou seja, longos trechos consecutivos de posições ocupadas, o que pode aumentar o custo das operações de busca (SOARES, 2022).

A implementação também inclui um sistema de entrada de dados com validação, utilizando a função `fgets` e a remoção do caractere de nova linha (`\n`) ao final das *strings*, a fim de evitar falhas nas comparações de texto. Além disso, o sistema conta com funcionalidades para salvar e carregar os dados em arquivos de texto, garantindo a persistência das informações entre diferentes execuções.

A escolha pela linguagem C se deve ao fato de ela proporcionar um controle direto sobre o gerenciamento de memória e permitir uma compreensão mais clara do funcionamento das estruturas de dados. Como as operações em tabelas *hash*, como inserção, busca e remoção, costumam apresentar tempo de execução constante, o sistema se mostra eficiente mesmo com um número elevado de palavras. Já a sondagem linear foi adotada por sua simplicidade de implementação.

4 Estrutura do Programa

A implementação do dicionário digital com tabela *hash* foi organizada em módulos, cada um dedicado a uma funcionalidade específica, o que facilita tanto a manutenção quanto a ampliação futura do código. A seguir, descrevemos os principais componentes do programa, correlacionando-os diretamente com trechos do código-fonte.

4.1 Structs e Definições

No começo do programa, são definidas três constantes: `size_word`, `size_mean` e `size_hash`. Essas constantes determinam, respectivamente, os limites máximos para o tamanho das palavras, dos significados e da tabela *hash* (Código 1).

```
1 #define size_word 50
2 #define size_mean 100
3 #define size_hash 19
```

Listing 1: Definição de constantes

A estrutura principal do programa é a `struct palavra`, que agrupa uma palavra e seu respectivo significado, através dos campos *word* e *mean*. Essa organização facilita o armazenamento conjunto das informações relacionadas (Código 2).

```c
typedef struct palavra{
    char word[size_word];
    char mean[size_mean];
} palavra;

palavra hash_table[size_hash];
```

Listing 2: Estrutura da palavra e vetor da tabela *hash*

O vetor `hash_table` representa a tabela *hash* com tamanho fixo de 19 posições, onde cada índice armazena um par palavra-significado. Cada posição funciona como uma entrada do dicionário digital, permitindo o acesso rápido aos dados por meio da função de h*ashing* (Código 2).

4.2 Função de *Hash*

A função de *hash* utilizada no projeto é denominada `Hash_String` e tem como objetivo converter um char (elementos que compoem a palavra a ser cadastrada) em um valor numérico inteiro (Código 3). Isso é feito por meio da soma dos valores ASCII de cada caractere da palavra.

```c
int Hash_String(const char *word){
    int hash = 0;
    for (int i = 0; word[i] != '\0'; i++)
        hash += (int)word[i];
    return hash;
}
```

Listing 3: Função de *hash* que soma valores ASCII das letras

Por exemplo, ao processar a palavra ”amor”, a função `Hash_String` soma os valores ASCII de cada caractere: 97 (`a`), 109 (`m`), 111 (`o`) e 114 (`r`), totalizando 431. Palavras que são anagramas, como ”amor” e ”roma”, possuem os mesmos caracteres em ordens diferentes, resultando na mesma soma ASCII.

No entanto, esse número não pode ser utilizado diretamente como índice da tabela *hash*, pois pode ultrapassar seu tamanho. Para isso, utiliza-se a função `MakeHashCode`, que aplica a operação de módulo para garantir que o índice gerado esteja dentro dos limites da tabela (de 0 a `size_hash - 1`) (Código 4).

```c
int MakeHashCode(int chave){
    return (chave % size_hash);
}
```

Listing 4: Função MakeHashCode

Dessa forma, ao passar a soma 431 para a função `MakeHashCode`, temos: $431 \mod 19 = 13$. Assim, tanto a palavra ”amor” quanto ”roma” serão armazenadas na posição 13 da tabela *hash*, o que pode causar uma colisão. Esse tipo de colisão, comum entre palavras que possuem os mesmos caracteres rearranjados, será tratado por meio da técnica de sondagem linear, a qual será detalhada posteriormente.

4.3 Inserção de Palavras no Dicionário

A função `Insert` é responsável por adicionar novos elementos ao dicionário (Código 5). Seu funcionamento inicia-se com a solicitação de uma palavra e de seu respectivo significado ao usuário. Para isso, a função `GetInputValue` é chamada, sendo responsável por interagir com o usuário e capturar os dados inseridos via teclado.

Em seguida, o programa aplica a função `Hash_String`, que converte a palavra em um valor numérico com base na soma dos códigos ASCII de seus caracteres. Esse valor é então passado à função `MakeHashCode`, que aplica a operação de módulo com o tamanho da tabela, resultando no índice onde a palavra deve ser armazenada.

Durante a inserção, pode ocorrer de o índice calculado já estar ocupado por outra palavra. Isso acontece quando diferentes palavras geram o mesmo valor de *hash*, resultando em uma colisão. Para lidar com esse problema, o programa utiliza a técnica de sondagem linear, que consiste em verificar sequencialmente as posições seguintes da tabela, aplicando a fórmula `indice = (indice + 1) % size_hash`, até encontrar uma célula vazia. A cada colisão detectada, a função `mens('c')`[1] é chamada, alertando o usuário sobre a colisão.

Se uma posição livre for encontrada, a palavra e seu significado são armazenados na tabela *hash*, e uma mensagem de sucesso é exibida. Caso todas as posições sejam verificadas sem sucesso, o programa informa que a tabela está cheia.

```
1 void Insert(){
2     palavra palav = GetInputValue();
3     int hash = Hash_String(palav.word);
4     int indice = MakeHashCode(hash);
5 
6     int tentativas = 0;
7     while (tentativas < size_hash) {
```

[1]Função auxiliar que imprime mensagens padronizadas ao usuário conforme o parâmetro recebido. As mensagens indicam o status da operação, como sucesso (’s’), falha (’f’), ausência de itens (’e’) ou ocorrência de colisão (’c’).

```
8           if (strcmp(hash_table[indice].word, "") == 0) {
9               hash_table[indice] = palav;
10              mens('s');
11              return;
12          } else {
13              mens('c');
14              indice = (indice + 1) % size_hash;
15              tentativas++;
16          }
17      }
18      mens('f');
19  }
```

Listing 5: Função Insert

4.4 Busca de Palavras no Dicionário

A função `Search` permite ao usuário consultar palavras já cadastradas no dicionário digital (Código 6). O processo de busca segue a mesma lógica utilizada na operação de inserção, ou seja, a função `Hash_String` transforma a palavra em um valor numérico e a função `MakeHashCode` calcula o índice inicial correspondente na tabela *hash*.

Após a leitura da palavra fornecida pelo usuário, o programa verifica se ela está armazenada na posição calculada. Caso contrário, aplica-se a técnica de sondagem linear para tratar possíveis colisões, verificando sequencialmente as posições seguintes com a fórmula `indice = (indice + 1) % size_hash`, até encontrar a palavra ou percorrer toda a tabela.

Se a palavra for encontrada, seu significado é exibido na tela. Caso contrário, é chamada a função `mens('e')`[1] para informar que a palavra não foi encontrada no dicionário.

```
1  void Search(){
2      char chave[size_word];
3      printf("# Pesquise pela palavra: \n>> ");
4      fgets(chave, size_word, stdin);
5      chave[strcspn(chave, "\n")] = 0;
6
7      int hash = Hash_String(chave);
8      int indice = MakeHashCode(hash);
9
10     int tentativas = 0;
```

```
11      while (tentativas < size_hash) {
12          if (strcmp(hash_table[indice].word, "") == 0) {
13              break;
14          }
15          if (strcmp(hash_table[indice].word, chave) == 0) {
16              printf("\n>> RESULTADO OBTIDO:\n\n");
17              printf("# %s\n", hash_table[indice].word);
18              printf("-> %s\n", hash_table[indice].mean);
19              printf("-----------------------------\n");
20              return;
21          }
22          indice = (indice + 1) % size_hash;
23          tentativas++;
24      }
25      mens('e');
26  }
```

Listing 6: Função para buscar uma palavra no dicionário

4.5 Remoção de Palavras do Dicionário

A função `Remove` tem como objetivo permitir que o usuário exclua uma palavra previamente cadastrada no dicionário digital (Código 7). Seu funcionamento baseia-se na mesma lógica apresentada anteriormente em inserção e busca, utilizando as funções `Hash_String` e `MakeHashCode` para calcular o índice inicial da palavra na tabela *hash* e aplicando a técnica de sondagem linear para lidar com possíveis colisões.

Após localizar a palavra, a remoção é efetuada sobrescrevendo seus campos com strings vazias, liberando o espaço para futuras inserções. O processo é concluído com a exibição de uma mensagem indicando o sucesso ou a falha da remoção, mantendo a integridade da estrutura da tabela.

```
1  void Remove(){
2      char chave[size_word];
3      printf("# Pesquise pela palavra: \n>> ");
4      fgets(chave, size_word, stdin);
5      chave[strcspn(chave, "\n")] = 0;
6
7      int hash = Hash_String(chave);
8      int indice = MakeHashCode(hash);
```

```
9
10      int tentativas = 0;
11      while (tentativas < size_hash) {
12          if (strcmp(hash_table[indice].word, "") == 0) {
13              break;
14          }
15          if (strcmp(hash_table[indice].word, chave) == 0) {
16              strcpy(hash_table[indice].word, "");
17              strcpy(hash_table[indice].mean, "");
18              mens('s');
19              return;
20          }
21          indice = (indice + 1) % size_hash;
22          tentativas++;
23      }
24      mens('e');
25  }
```

Listing 7: Função para remover uma palavra do dicionário

4.6 Exibição do Conteúdo da Tabela *Hash*

A função `PrintHash` é responsável por percorrer toda a tabela *hash* e exibir todas as palavras e seus significados armazenados (Código 8). Essa funcionalidade é necessária para fins de verificação, permitindo que o usuário visualize o conteúdo completo do dicionário digital.

A implementação consiste em um laço que percorre todas as posições do vetor da tabela *hash*. Para cada entrada que contenha uma palavra válida, ou seja, diferente de string vazia, a função imprime a palavra e seu significado.

```
1  void PrintHash(){
2      printf("\n%15s\n", "Palavras: ");
3      for(int i = 0; i < size_hash; i++){
4          if (strcmp(hash_table[i].word, "") != 0) {
5              printf("# %s\n", hash_table[i].word);
6              printf("-> %s\n", hash_table[i].mean);
7          }
8      }
9      printf("-----------------------------\n");
10 }
```

Listing 8: Função para exibir todo o conteúdo do dicionário

Essa visualização permite ao usuário acompanhar a organização interna da estrutura de dados e confirmar a correta inserção e remoção das informações.

4.7 Persistência dos Dados em Arquivo

O programa implementa um sistema de persistência de dados por meio de arquivos de texto, possibilitando a manutenção das informações armazenadas mesmo após o encerramento da aplicação. Esse recurso é fundamental para garantir que os dados se mantenham entre diferentes execuções do sistema.

Para isso, são utilizadas duas funções principais: `Load` e `Save`, as quais empregam as bibliotecas padrão da linguagem C para realizar operações de entrada e saída em arquivos.

- **Função Load**: é executada no início do programa e tem como objetivo carregar os dados salvos no arquivo `dicionario.txt`, memorizando os dados do dicionário. Caso o arquivo ainda não exista, como na primeira execução, a função simplesmente não realiza a leitura, permitindo que o programa inicialize normalmente com a tabela *hash* vazia (Código 9).

```
1  void Load(){
2      FILE *arquivo = fopen("dicionario.txt", "r");
3      if (!arquivo){
4          return;
5      }
6
7      char word[size_word];
8      char mean[size_mean];
9
10     while (fgets(word, size_word, arquivo) != NULL &&
11            fgets(mean, size_mean, arquivo) != NULL) {
12         word[strcspn(word, "\n")] = 0;
13         mean[strcspn(mean, "\n")] = 0;
14
15         int hash = Hash_String(word);
16         int indice = MakeHashCode(hash);
17         int tentativas = 0;
18
19         while (tentativas < size_hash) {
```

```
20              if (strcmp(hash_table[indice].word, "") == 0) {
21                  strcpy(hash_table[indice].word, word);
22                  strcpy(hash_table[indice].mean, mean);
23                  break;
24              }
25              indice = (indice + 1) % size_hash;
26              tentativas++;
27          }
28      }
29      fclose(arquivo);
30  }
```

Listing 9: Função Load

- **Função Save**: percorre toda a tabela *hash* e grava, linha por linha, cada palavra e seu respectivo significado no arquivo `dicionario.txt`. Dessa forma, o coteúdo do dicionário é salvo (Código 10).

```
1  void Save(){
2      FILE *arquivo = fopen("dicionario.txt", "w");
3      if(!arquivo){
4          mens('f');
5          exit(1);
6      }
7      for(int i = 0; i < size_hash; i++){
8          fprintf(arquivo, "%s\n", hash_table[i].word);
9          fprintf(arquivo, "%s\n", hash_table[i].mean);
10     }
11     fclose(arquivo);
12 }
```

Listing 10: Função Save

5 Testes Realizados

A fim de assegurar a confiabilidade e eficiência do sistema, foram conduzidos testes em diferentes cenários, envolvendo inserção, busca e remoção de palavras, incluindo situações com colisões, buscas por palavras inexistentes e verificação da persistência dos dados em arquivo.

Inicialmente, foi inserida a palavra “amor”, com seu respectivo significado, conforme ilustrado na Figura 1. Em seguida, adicionaram-se as palavras “roma” e “gato”. A exibição da tabela após essas inserções confirmou que os dados foram armazenados corretamente, inclusive nos casos de colisão.

As palavras “amor” e “roma” geraram colisão ao serem mapeadas para o mesmo índice pela função de *hash*. No entanto, a sondagem linear permitiu a realocação correta da palavra “roma” para a próxima posição disponível, conforme evidenciado na Figura 2. A Figura 3 mostra a exibição completa da tabela após a inserção das três palavras: “amor”, “roma” e “gato”.

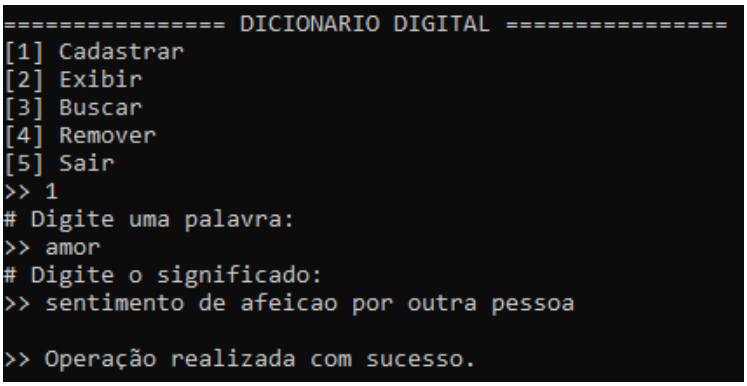

Figura 1: Inserção da palavra “amor”.

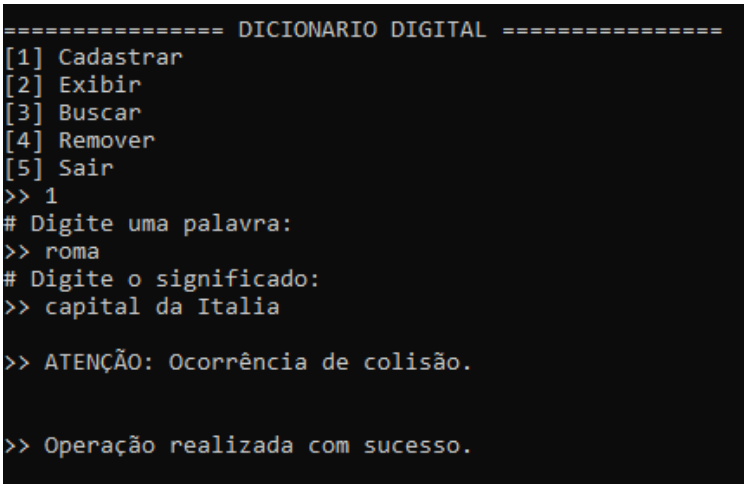

Figura 2: Colisão com a palavra “roma”, armazenada corretamente em outra posição.

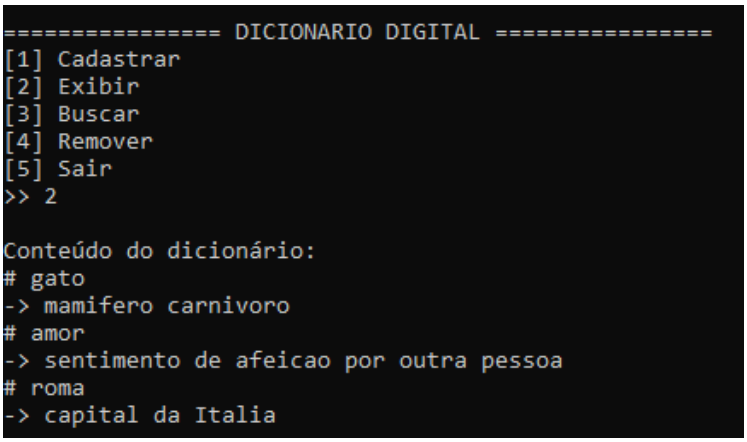

Figura 3: Exibição das palavras “amor”, “roma” e “gato”.

Também foram realizados testes de busca das palavras previamente inseridas na tabela, e todas retornaram os resultados esperados. A busca demonstrou ser eficiente mesmo em situações com colisões. Como exemplo, a Figura 4 ilustra a busca bem-sucedida pela palavra “gato”.

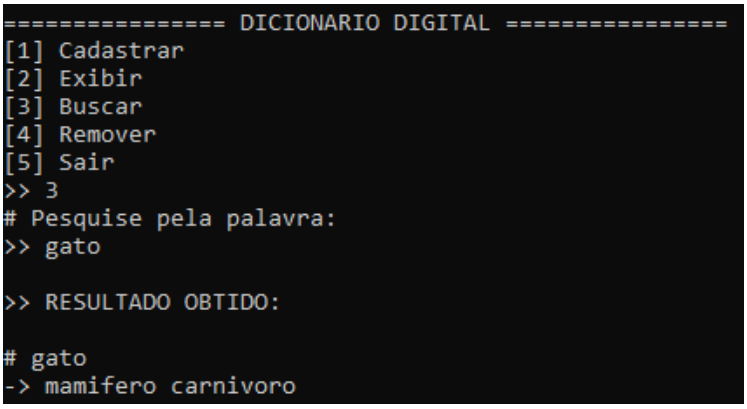

Figura 4: Busca pela palavra “gato” na tabela *hash*.

A palavra “gato” foi removida com sucesso da tabela *hash*. Como ilustrado nas Figuras 5 e 6, o sistema executou corretamente a exclusão da palavra e confirmou sua ausência na busca da palavra após a remoção, como pode ser observado na Figura 6.

Além disso, a integridade dos demais dados foi preservada durante o processo de remoção, o que demonstra que o sistema gerencia adequadamente os dados e que a exclusão de um elemento não afeta os demais registros na estrutura de dados.

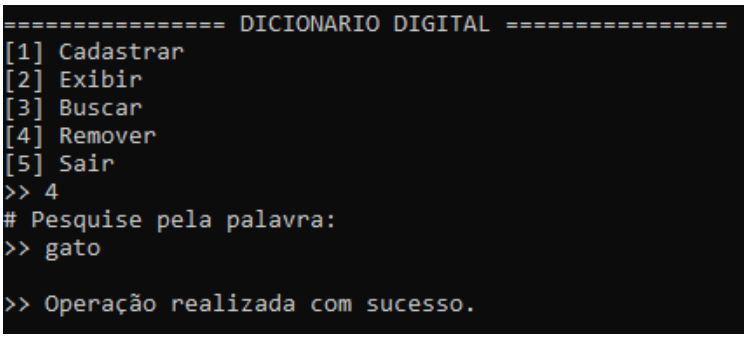

Figura 5: Remoção da palavra “gato” do dicionário.

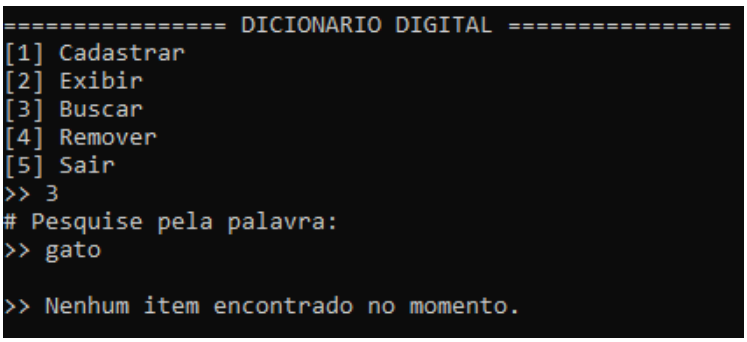

Figura 6: Busca da palavra “gato” após a sua remoção.

Após encerrar e reiniciar o programa, os dados armazenados foram carregados corretamente a partir do arquivo `dicionario.txt`, demonstrando que a persistência em arquivo foi bem-sucedida. As palavras “amor” e “roma” continuaram presentes na tabela *hash*, conforme pode ser observado na Figura 7.

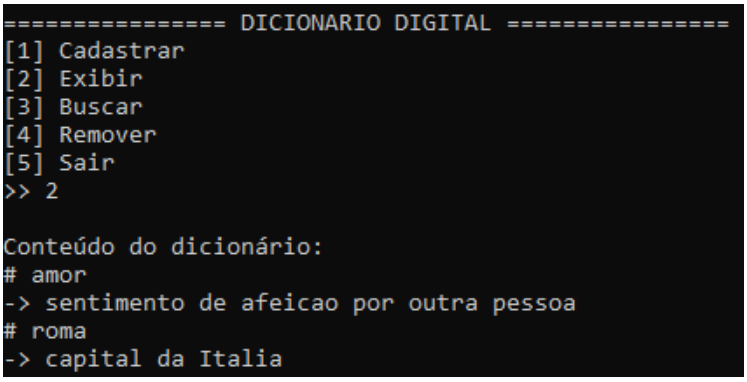

Figura 7: Tabela *hash* após reinício do programa, com preservação das palavras “amor” e “roma”.

Dessa forma, as operações testadas demonstraram que o sistema funciona conforme o esperado em diferentes situações.

6 Resultados Obtidos

O programa implementado demonstrou eficácia nas operações de inserção, busca, remoção e exibição de palavras e de seus respectivos significados em uma tabela *hash* com sondagem linear. Todas as palavras inseridas foram corretamente armazenadas, inclusive nos casos em que ocorreram colisões, evidenciando o funcionamento adequado da técnica de tratamento de colisões adotada.

A função de *hash*, baseada na soma dos códigos ASCII dos caracteres, gerou uma distribuição satisfatória dos dados na tabela, sendo capaz de lidar com entradas compostas por caracteres rearranjados, sem comprometer a integridade das informações. As colisões observadas foram resolvidas corretamente por meio da sondagem linear, como evidenciado na Figura 2, que exemplifica o caso das inserções das palavras “amor” e “roma”.

Os testes também mostraram que as operações de busca retornaram corretamente os significados das palavras previamente inseridas, enquanto as buscas por termos inexistentes resultaram em mensagens informativas ao usuário sobre o ocorrido. A remoção de palavras funcionou como esperado, garantindo que os elementos excluídos não fossem mais encontrados na tabela. A funcionalidade de exibição se mostrou útil para apresentar ao usuário todo o conteúdo armazenado na tabela, permitindo uma visão geral dos dados e facilitando a verificação da organização do dicionário.

Por fim, verificou-se que a persistência dos dados foi bem-sucedida. Após o encerramento do sistema, todas as palavras previamente armazenadas foram recuperadas a partir do arquivo de texto, mantendo-se intactas durante as execuções subsequentes. Esses resultados confirmam que o problema proposto foi solucionado, evidenciando a eficiência da estrutura desenvolvida para o gerenciamento de dados no dicionário digital.

7 Dificuldades e Desafios

Durante o desenvolvimento do sistema, diversos desafios foram enfrentados, tanto no nível lógico quanto na implementação do código em linguagem C. Um dos principais obstáculos foi o correto tratamento das colisões na tabela *hash*, especialmente em cenários com maior número de dados. A aplicação da sondagem linear exigiu atenção quanto à lógica de realocação e verificação de espaços disponíveis, evitando sobreposição ou perda de informações.

A persistência dos dados também se mostrou desafiadora. Foi necessário garantir que as palavras e seus respectivos significados, inseridos pelo usuário, fossem corretamente salvos em arquivo de texto e, posteriormente, recuperados de forma íntegra nas execuções subsequentes da aplicação.

Outro obstáculo recorrente envolveu o uso de cadeias de caracteres. Devido à ausência

de suporte nativo para strings dinâmicas em C, foi necessário utilizar com precisão funções como `fgets`, `strcmp` e `strcpy`, além de remover manualmente caracteres indesejados, como a quebra de linha gerada pela entrada padrão.

Por fim, além dos aspectos técnicos, buscou-se construir uma interface textual simples e funcional, com menus bem organizados e mensagens claras. Essa preocupação com a usabilidade contribuiu para que o sistema fosse intuitivo, mesmo diante das limitações da linguagem e do escopo do problema proposto.

8 Conclusão

A implementação do dicionário digital utilizando tabela *hash* com a técnica de sondagem linear demonstrou ser uma solução adequada para o armazenamento e gerenciamento de palavras e seus significados. A estrutura adotada mostrou-se capaz de organizar os dados de forma consistente, preservando a integridade das informações mesmo diante de colisões.

A funcionalidade de persistência, por meio do salvamento e carregamento dos dados em arquivo de texto, garantiu a continuidade das informações entre diferentes execuções da aplicação, contribuindo para o sucesso da solução proposta.

O sistema desenvolvido atendeu aos requisitos estabelecidos, oferecendo uma ferramenta prática e funcional que evidencia a aplicabilidade das tabelas *hash* em contextos reais. Dessa forma, o projeto reforça a importância e eficiência dessa estrutura na organização e gestão de dados textuais.

Referências

BENTO, L. M. S.; PEREIRA DE SÁ, V. G.; SZWARCFITER, J. L. Hashing na Solução de Problemas Atípicos. In: CONGRESSO Latino Iberoamericano de Investigação Operativa e Simpósio Brasileiro de Pesquisa Operacional. [S.l.: s.n.], 2012. Acesso em: 20 jul. 2025. Disponível em: <`http://www.din.uem.br/sbpo/sbpo2012/pdf/arq0449.pdf`>.

SOARES, Antônio Michael Farias. **Implementação e avaliação de banco de dados chave-valor com aprendizagem de índice**. [S.l.: s.n.], 2022. Disponível em: <`https://repositorio.ufc.br/bitstream/riufc/65034/1/2022_tcc_amfsoares.pdf`>. Acesso em: 18 jul. 2025.