Documentação AED – Trabalho Prático II

Izabela Andrade, Thalita Marques

Curso de Bacharelado em Sistemas de Informação – Pontifícia Universidade Católica de Minas Gerais (PUC Minas) – Campus de Contagem
32265-450 – Contagem – MG– Brasil

izabelaima@gmail.com, thalita.sylvia@gmail.com

1. Introdução

A proposta do presente trabalho consiste em desenvolver um programa em C# que implemente a Estrutura de Dados Tabela Hash. A estrutura será usada para armazenar registros de todos os estados brasileiros de forma que facilite, posteriormente, a busca desses elementos, visto que com uma boa função hash pode-se evitar comparações que não encontram o elemento procurado. À estas comparações dá-se o nome de "colisões". Uma Tabela Hash bem elaborada tem tamanho e função hash estratégicas e assim contribui para algoritmos de busca e inserção que custem menos, principalmente em questão de tempo. O sucesso da implementação é então o equilíbrio entre os gastos de tempo das execuções e o espaço da memória.

2. Funcionamento

Definido pela proposta de implementação, o usuário deve escolher, em tempo de execução, qual será o tamanho da Tabela Hash a ser criada, bem como o tipo de Tratamento de Colisões utilizado: Lista Encadeada ou Endereçamento Aberto. Não lhe é permitido informar um número negativo ou nulo para o tamanho da tabela, no caso do Endereçamento Aberto inclusive, não lhe é permitido criar uma tabela menor que o número de estados no Brasil (27), pois, já que conhecemos previamente as chaves a serem inseridas, podemos fazer uso deste controle tão útil. Os elementos são formados por dados de entrada vindos de um arquivo de texto chamado estados.txt logo após o usuário fornecer informações para a criação da tabela. Então o usuário tem a oportunidade de pesquisar por quantos estados quiser, sendo informado do número de colisões a cada busca por uma nova chave. Definido pela programação, inserção e busca têm mesmo número de colisões. O usuário também pode visualizar na tela como está a tabela e as posições em que se encontram cada chave (estado) ao final das pesquisas.

3. Elementos e Chaves

Os elementos a serem inseridos na Tabela Hash são instâncias da classe Estado. A classe representa um estado do Brasil tendo como atributos seu nome (string), capital (string), região (string) e quantidade de municípios (int). Além desses campos, a classe conta também com o método construtor, que recebe os dados como parâmetros para inicializa-los e com o método imprime, que mostra na tela o conteúdo desses campos.

O campo nome do estado é considerado a chave, que através do método função hash, tem sempre uma posição específica para endereçamento de todo o elemento (instância de estado) na Tabela Hash.

4. Estruturas de Dados

No caso do usuário escolher Listas Encadeadas para tratar colisões, criamos a tabela hash como um vetor de listas encadeadas. Já no caso de Endereçamento Aberto, utilizamos apenas um vetor de objetos do tipo Estado. Porém, para que este último funcione corretamente, precisa ter mais posições do que o primeiro.

4.1. Listas Encadeadas

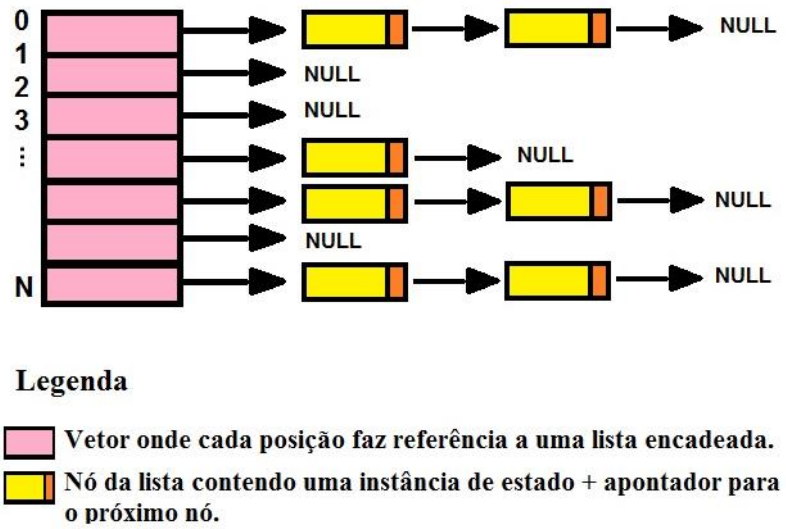

Figura 1. Esquema da Estrutura de Dados utilizada para implementação da Tabela Hash no caso de Listas Encadeadas para tratamento de colisões.

4.2. Endereçamento Aberto

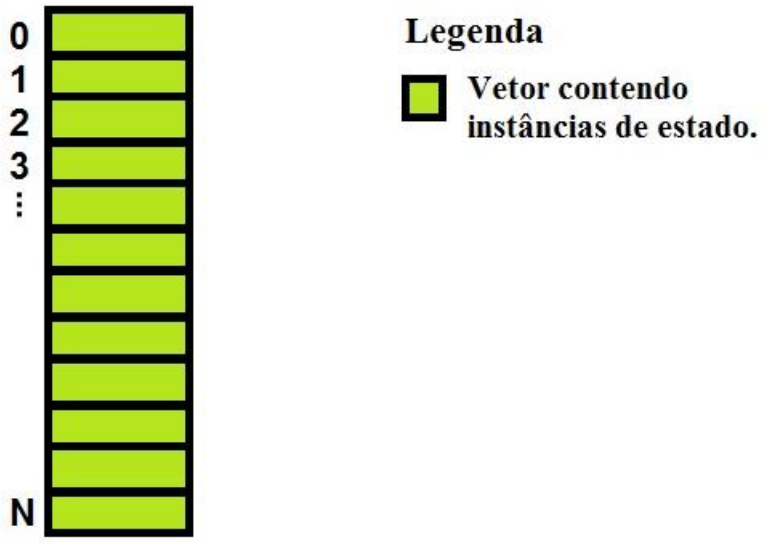

Figura 2. Esquema da Estrutura de Dados utilizada para implementação da Tabela Hash no caso de Endereçamento Aberto para tratamento de colisões.

5. Herança

Como o tipo de Tratamento de Colisões altera totalmente a forma como a Tabela Hash funciona, precisaríamos de duas classes de Tabela Hash diferentes. Porém elas não deixariam de ter muito em comum, por isso resolvemos usar Herança, o que reduziu consideravelmente o código além de tê-lo deixado bem mais modularizado e sem redundâncias.

Criamos uma Classe Abstrata Hash contendo o atributo (int)tamanho e o método funcaoHash(), já que estes seriam iguais para as duas situações possíveis, e os métodos que seriam diferentes tiveram apenas suas assinaturas definidas e foram declarados como abstratos, assim como a classe. A maior vantagem da herança foi poder declarar a tabela como sendo do tipo Hash, e depois poderia ser atribuída a ela tanto uma instância de HashLE quanto de HashEA.

```
abstract class Hash
{
    public int tamanho;

    public int funcaoHash(string chave)
    {
        int soma = 0;

        for (int i = 0; i < chave.Length; i++)
        {
            soma += (chave[i] * i);
        }

        return (soma % tamanho);
    }

    public abstract void InsereNaTabela(Estado estado);

    public abstract void procuraEstado(string nome);

    public abstract void imprimir();
}
```

6. Função Hash

Tanto para inserir quanto para buscar por um elemento/chave na tabela Hash é utilizado uma função que retorna uma posição ou endereço na tabela. Essa função é definida pelo programador e neste trabalho escolhemos multiplicar a posição alfabética de cada letra da chave, sendo esta do tipo string (nome do estado), pela posição da letra dentro da própria string. Para cada caractere na string, repetimos o procedimento somando e acumulando essa soma. No final da função hash retornamos o módulo da soma com o tamanho da tabela. Esse retorno é usado como posição da tabela. O método pertence à classe abstrata citada acima.

```
public int funcaoHash(string chave)
{
    int soma = 0; //acumula para toda iteração do for a seguir

    for (int i = 0; i < chave.Length; i++)
    {
        soma += (chave[i] * i); //multiplica a letra da chave pela sua
                                //posição
    }

    return (soma % tamanho); //valor acumulado módulo tamanho da tabela
}
```

7. HashLE

Como citado, a classe HashLE é filha da classe abstrata Hash e implementa alguns métodos abstratos de maneira particular. São estes: insereNaTabela(Estado estado), procuraEstado(string nome) e imprimir().

7.1. insereNaTabela(Estado estado)

```
public override void InsereNaTabela(Estado estado)
{
    int pos = funcaoHash(estado.nome);

    registros[pos].insereFim(estado);

    /*  Método utiliza o método funcaoHash para saber a posição
     *  Logo após, insere no fim da Lista de colisões.
     *  Cada registro da tabela é uma lista de elementos
     *  com mesmo endereçamento
     */
}
```

7.2. procuraEstado(string nome)

```
//Método recebe o nome de um estado e o procura na Tabela Hash.
public override void procuraEstado(string nome)
{
    Node aux;
    int numColisoes = 0, pos;

    // Utilizando a função hash o método sabe em qual endereço
    //o estado supostamente está
    pos = funcaoHash(nome);
    aux = registros[pos].inicio;
```

```
            //Percorre toda a lista de colisões no endereço encontrado,
            //contando também quantas foram as colisões
            while ((nome != aux.estado.nome) && (aux.next != null))
            {
                numColisoes++;
                aux = aux.next;
            }

            if (nome == aux.estado.nome)//se encontra o estado certo na tabela
            {
                //mostra na tabela o número de colisões
                //e os dados completos do estado
                Console.WriteLine("Chave com {0} colisões", numColisoes);
                aux.estado.imprimir();
            }
            else//se a chave informada não está na tabela o usuário é avisado
            {
                Console.WriteLine("Chave {0} inexistente na tabela!", nome);
            }
        }
```

7.3. imprimir()

```
        public override void imprimir()//imprime toda a tabela
        {
            Node aux;
            Console.WriteLine("POS\t   CHAVES\n");

            //percorre todos os endereços
            for (int pos = 0; pos < registros.Length; pos++)
            {
                Console.Write(pos + "\t"); //imprime posição de cada endereço

                aux = registros[pos].inicio;

                //caso não tenham elementos em determinado endereço
                if (aux == null)
                {
                    Console.Write("  -");
                }
                else//caso tenham elementos no endereço
                {
                    do//percorre toda a lista de colisões imprimindo as chaves
                    {
                        Console.Write(aux.estado.nome + ",  ");
                        aux = aux.next;
                    } while (aux != null);
                }
                Console.WriteLine();
            }

        }
```

8. HashEA

A classe HashEA, assim como a anterior, também é filha da classe abstrata Hash e implementa os mesmos métodos abstratos de maneira particular.

8.1. insereNaTabela(Estado estado)

```
public override void InsereNaTabela(Estado estado)
{
    //Método utiliza o método funcaoHash para saber a posição
    int pos = funcaoHash(estado.nome);

    //se o endereço está ocupado, percorre os endereços até que
    //um esteja livre
    while(registros[pos] != null)
    {
        pos++;
        if(pos == registros.Length)//se chegar no fim,
        {
            pos = 0;//volta no início
        }
        //não há possibilidade de infinity looping pois
        //o tamanho da tabela é adequado (validado ao receber do usuário)
    }
    registros[pos] = estado;

}
```

8.2. procuraEstado(string nome)

```
public override void procuraEstado(string nome)
{
    int pos, posInicial, numColisoes = 0;
    bool existe = true;

    // Utilizando a função hash o método sabe em qual
    //endereço o estado supostamente está
    pos = funcaoHash(nome);

    posInicial = pos;//anula possibilidade de infinity looping,
                     //no caso de busca por chave inexistente

    //percorre os endereçamentos
    while((registros[pos] != null)&&
          (registros[pos].nome != nome)&&
          (existe == true))
    {
        pos++;
        numColisoes++;
        if(pos == registros.Length)//se chegar ao fim,
        {
            pos = 0;//volta ao início
        }
        else if(pos == posInicial)
        {
            //se percorreu todos os endereçamentos sem encontrar a chave
            Console.WriteLine("Chave {0} inexistente na tabela!", nome);
            existe = false;
        }
    }
```

```
        if(registros[pos] == null)
        {
            Console.WriteLine("Chave {0} inexistente na tabela!", nome);
            existe = false;
        }
        if (existe)//localizou a chave (registros[pos].nome == nome)
        {
            Console.WriteLine("Chave com {0} colisões", numColisoes);
            registros[pos].imprimir();
        }
    }
```

8.3. imprimir()

```
    public override void imprimir()//imprime toda a tabela
    {
        Console.WriteLine("POS\t CHAVES\t                          h(x)\n");

        //percorre endereçamentos
        for (int pos = 0; pos < registros.Length; pos++)
        {
            Console.Write(pos + "\t");//imprime a posição
            if(registros[pos] == null)//caso não tenha chave no endereço
            {
                Console.Write("  -");
            }
            else//caso tenha uma chave
            {
                Console.Write(registros[pos].nome);
               //imprime a chave (nome do estado)

                for (int i = 0; i < (30-registros[pos].nome.Length); i++)
                {
                    //este laço garante um alinhamento para exibir "h(x)="
                    Console.Write(" ");
                }

                //imprime seu resultado de hashing primario
                Console.Write("h(x)=" + funcaoHash(registros[pos].nome));
            }
            Console.WriteLine();
        }
    }
```

9. Classes Lista e Node

A classe Lista é o modelo da Estrutura de Dados de Lista Encadeada utilizada para armazenar as chaves de mesmo resultado de função hash, ou seja, chaves que colidiram no memento de inserção (no caso do Tratamento de Colisões ser por Listas Encadeadas). A classe conta com três métodos, sendo eles o construtor, o vazia (verifica se a lista já tem elementos inseridos) e o insereFim (que recebe uma instância de estado e o insere no fim da lista).

A classe Node é o modelo de cada nó na estrutura de Lista Encadeada. Cada nó não nulo guarda uma instância de um estado e um ponteiro para o próximo nó. De método, conta apenas com o seu construtor.

10. Classe Program

Na Classe Program temos dois métodos sendo utilizados, buscarDados e o Main, onde começa a execução do programa.

10.1. levarDados()

O método leverDados deixamos comentado no código, pois não é usado durante a execução, mas o chamamos uma vez para criar um arquivo de texto com os dados retirados da internet.

```
////método utilizado para escrever os dados no arquivo de texto, não é mais
utilizado na execução
//public static void levarDados()
//{
//    FileStream arq = new FileStream("estados.txt", FileMode.OpenOrCreate);
//    StreamWriter sw = new StreamWriter(arq);
//    //string nome;//campo chave
//    //string capital;
//    //string regiao;
//    //int quantMunicipios;
//    sw.WriteLine("ACRE|RIO BRANCO|NORTE|22");
//    sw.WriteLine("ALAGOAS|MACEIO|NORDESTE|102");
//    sw.WriteLine("AMAPA|MACAPA|NORTE|16");
//    sw.WriteLine("AMAZONAS|MANAUS|NORTE|62");
//    sw.WriteLine("BAHIA|SALVADOR|NORDESTE|417");
//    sw.WriteLine("CEARA|FORTALEZA|NORDESTE|184");
//    sw.WriteLine("DISTRITO FEDERAL|BRASILIA|CENTRO-OESTE|1");
//    sw.WriteLine("ESPIRITO SANTO|VITORIA|SUDESTE|78");
//    sw.WriteLine("GOIAS|GOIANIA|CENTRO-OESTE|246");
//    sw.WriteLine("MARANHAO|SAO LUIS|NORDESTE|217");
//    sw.WriteLine("MATO GROSSO|CUIABA|CENTRO-OESTE|141");
//    sw.WriteLine("MATO GROSSO DO SUL|CAMPO GRANDE|CENTRO-OESTE|79");
//    sw.WriteLine("MINAS GERAIS|BELO HORIZONTE|SUDESTE|853");
//    sw.WriteLine("PARA|BELEM|NORTE|144");
//    sw.WriteLine("PARAIBA|JOAO PESSOA|NORDESTE|223");
//    sw.WriteLine("PARANA|CURITIBA|SUL|399");
//    sw.WriteLine("PERNAMBUCO|RECIFE|NORDESTE|185");
//    sw.WriteLine("PIAUI|TERESINA|NORDESTE|224");
//    sw.WriteLine("RIO DE JANEIRO|RIO DE JANEIRO|SUDESTE|92");
//    sw.WriteLine("RIO GRANDE DO NORTE|NATAL|NORDESTE|167");
//    sw.WriteLine("RIO GRANDE DO SUL|PORTO ALEGRE|SUL|496");
//    sw.WriteLine("RONDONIA|PORTO VELHO|NORTE|52");
//    sw.WriteLine("RORAIMA|BOA VISTA|NORTE|15");
//    sw.WriteLine("SANTA CATARINA|FLORIANOPOLIS|SUL|293");
//    sw.WriteLine("SAO PAULO|SAO PAULO|SUDESTE|645");
//    sw.WriteLine("SERGIPE|ARACAJU|NORDESTE|75");
//    sw.WriteLine("TOCANTINS|PALMAS|NORTE|139");
//    sw.Close();
//}
```

10.2. buscarDados(Hash tabela)

O método buscarDados recebe como parâmetro uma Tabela Hash qualquer, onde vai depositar os dados. O arquivo então é lido linha a linha, inicializando uma nova instância de Estado com os dados da linha, e já insere na tabela (parâmetro) o estado (objeto criado). O método a ser usado para inserir na tabela vai depender de que tipo de tabela ela é, HashLE ou HashEA. Porém a execução sabe qual chamar sem que precisemos nos preocupar com isso, graças à Herança na Orientação a Objetos.

```
public static void buscarDados(Hash tabela)
{
    string[] v;
    string linha;

    FileStream file = new FileStream("estados.txt", FileMode.Open);
    //objeto representa o arquivo

    StreamReader sr = new StreamReader(file);
    //objeto faz a leitura no arquivo especificado

    do//estrutura que garante a leitura do arquivo linha a linha
    {
        linha = sr.ReadLine();
        if (linha != null)
        {
            v = linha.Split('|');

            //os dados de uma linha completa no arquivo
            //se tornam uma nova instância de Estado
            //que já é inserida na tabela
            tabela.InsereNaTabela(new Estado(v[0], v[1], v[2], int.Parse(v[3])));
        }
    } while (linha != null);

    sr.Close();

}
```

10.3. Main

No método Main temos as entradas do usuário de tipo de Tratamento de Colisões e Tamanho da tabela sendo inseridas e validadas. O tamanho precisa ser um número positivo, sendo que se o Tratamento de Colisões for por Endereçamento Aberto, o tamanho precisa ser maior do que o número total e chaves, que é 27 (estados do Brasil).

A variável tabelaHash é declarada logo no início. Sendo do tipo Hash, pode receber uma instância tanto de HashLE quanto de HashEA.

Depois de criada a tabela, o usuário pode pesquisar quantos estados desejar até responder “S” quando for perguntado se deseja sair. O usuário é informado do número de colisões na busca da chave. Após sair da pesquisa de chaves, há a possibilidade de visualizar todos os elementos na tabela hash assim como estão dispostos na mesma. Essa visualização ocorre através do método abstrato imprimir, implementado nas filhas da classe Hash.

11. Testes

Os testes foram divididos de acordo com as duas possibilidades de tratamento de colisões: com Listas Encadeadas e com Endereçamento Aberto, já que os algoritmos seriam diferentes de um tipo de tratamento para outro. Para todos os testes, as chaves foram inseridas em ordem alfabética, que era a ordem em que estavam dispostas no arquivo de texto de entrada. A cada linha lida do arquivo de texto era instanciado um estado que era inserido na Tabela Hash utilizando seu nome como chave. A imagem a seguir mostra os dados no arquivo.

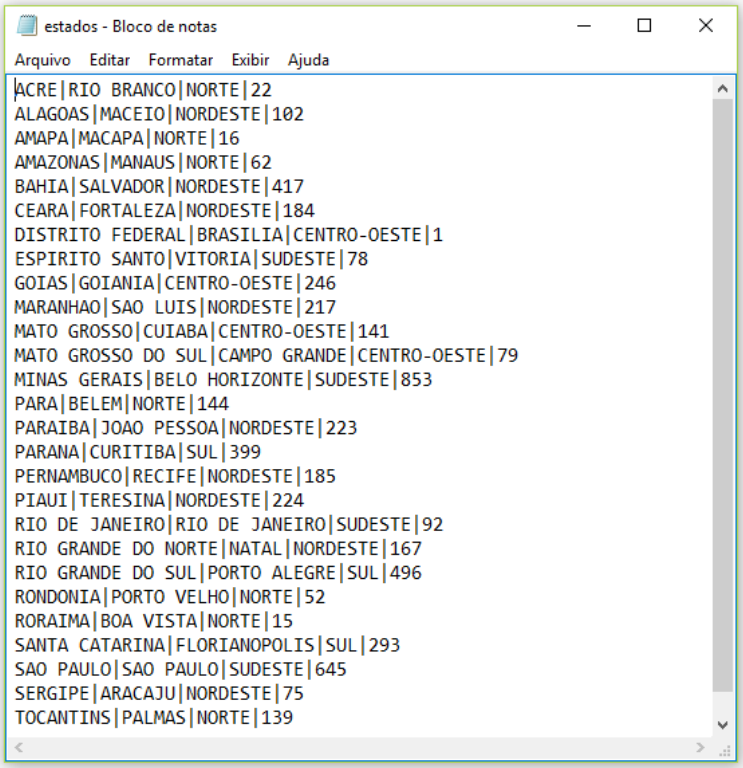

Figura 3. Arquivo txt com dados de entrada.

11.1. Com Listas Encadeadas

O usuário pode pesquisar por quantas chaves desejar e, posteriormente, pode visualizar como estava a tabela hash (apenas para que entenda melhor seu funcionamento, mesmo durante a execução). A seguir, testes de criação da tabela com validação do tamanho escolhido, que deve ser maior que 0 (zero); teste de busca por chaves mostrando número de colisões em cada busca ou mensagem informando caso a chave não exista na tabela; teste de visualização da tabela. Foi criada tabela de tamanho 11.

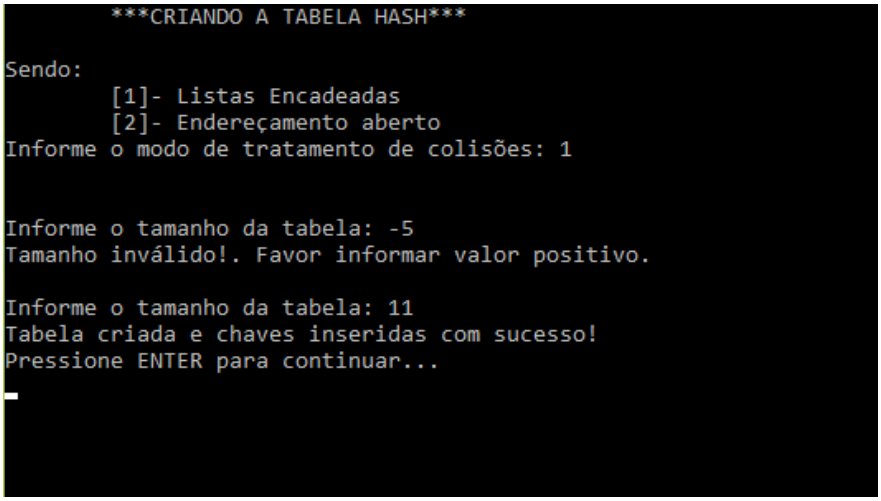

Figura 4. Teste de criação da Tabela Hash com Listas Encadeadas para tratamento de colisões e validação do tamanho.

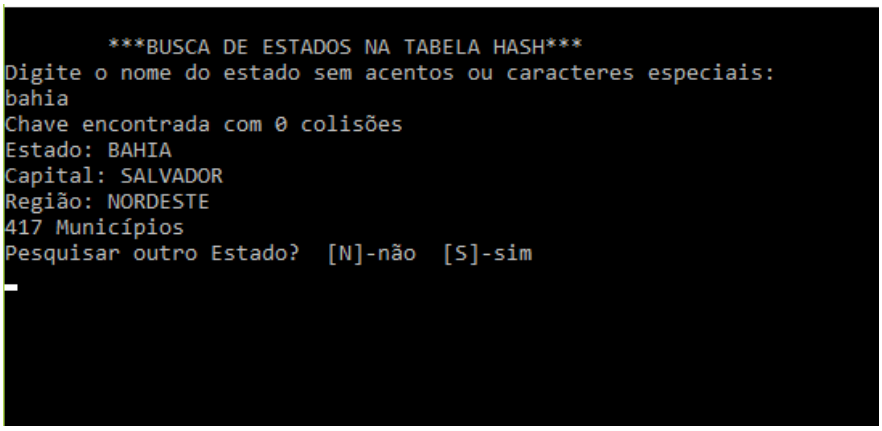

Figura 5. Exemplo de busca de chave sem colisões.

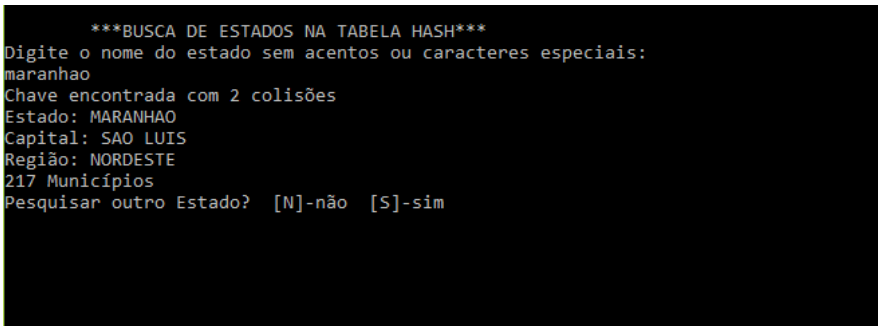

Figura 6. Exemplo de busca de chave no final de uma lista.

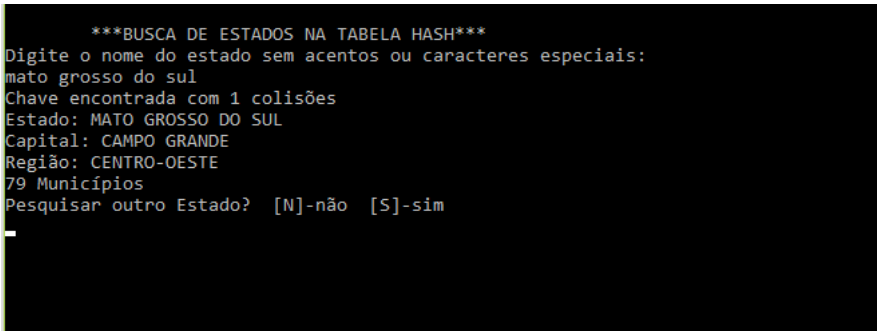

Figura 7. Exemplo de busca de uma chave no meio de uma lista.

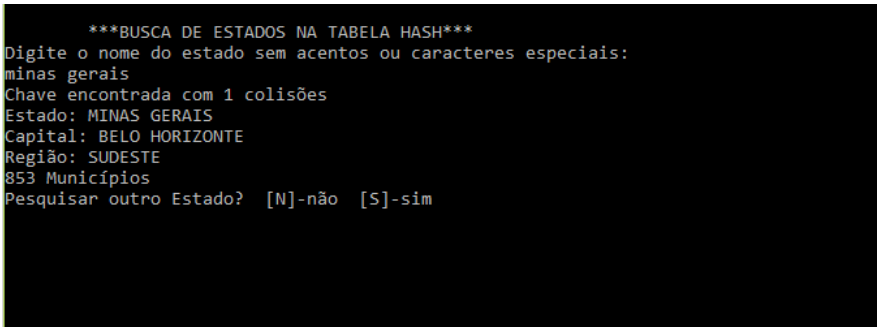

Figura 8. Exemplo de busca de uma chave no meio de uma lista.

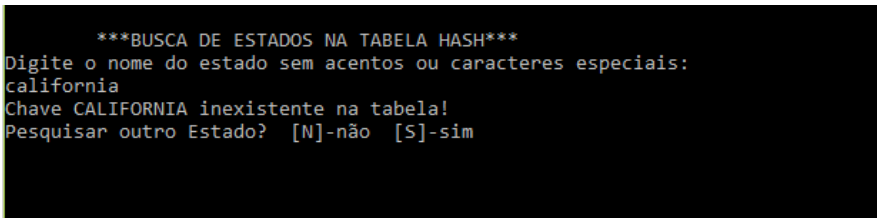

Figura 9. Exemplo de busca por chave inexistente.

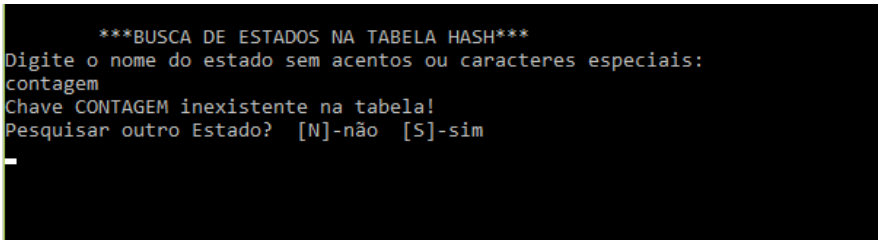

Figura 10. Exemplo de busca por chave inexistente.

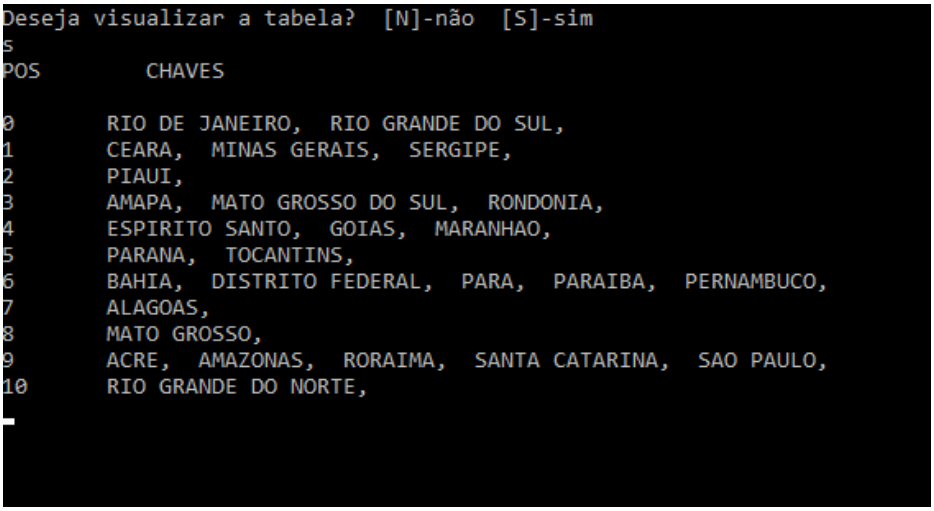

Figura 11. Teste de visualização da tabela e suas chaves.

11.2. Com Endereçamento Aberto

Da mesma maneira, o usuário pode pesquisar por quantas chaves desejar e, posteriormente, pode visualizar como estava a tabela hash (apenas para que entenda melhor seu funcionamento, mesmo durante a execução). A seguir, testes de criação da tabela com validação do tamanho escolhido, que deve ser maior que a tabela hash para que comporte todos os elementos; teste de busca por chaves mostrando número de colisões em cada busca ou mensagem informando caso a chave não exista na tabela; teste de visualização da tabela bem como o valor da função hash para cada chave, apenas para que se facilite o entendimento das colisões. Foi criada tabela de tamanho 31.

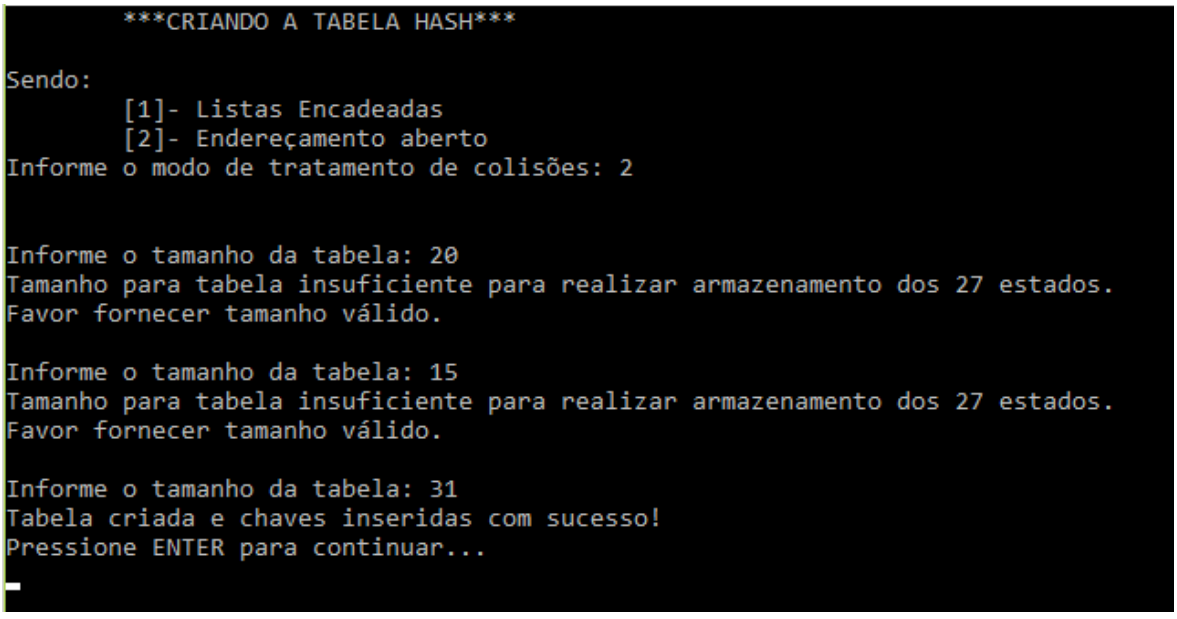

Figura 12. Teste de criação da tabela com Endereçamento Aberto para tratamento de colisões com validação do tamanho.

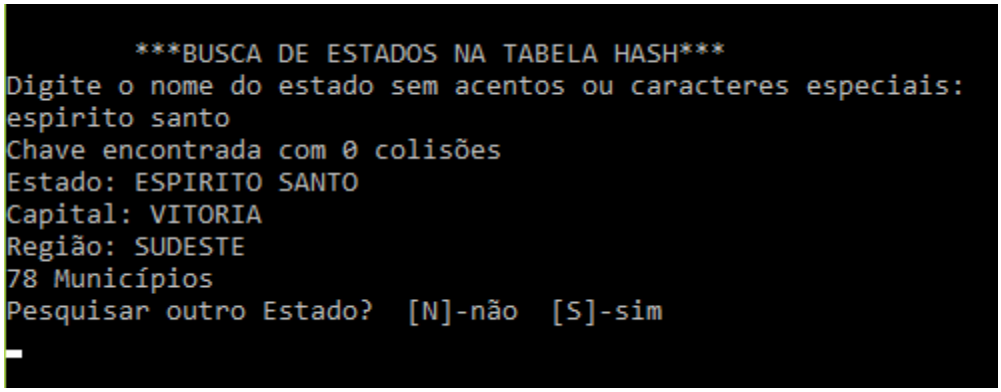

Figura 13. Teste de busca por chave ESPIRITO SANTO.

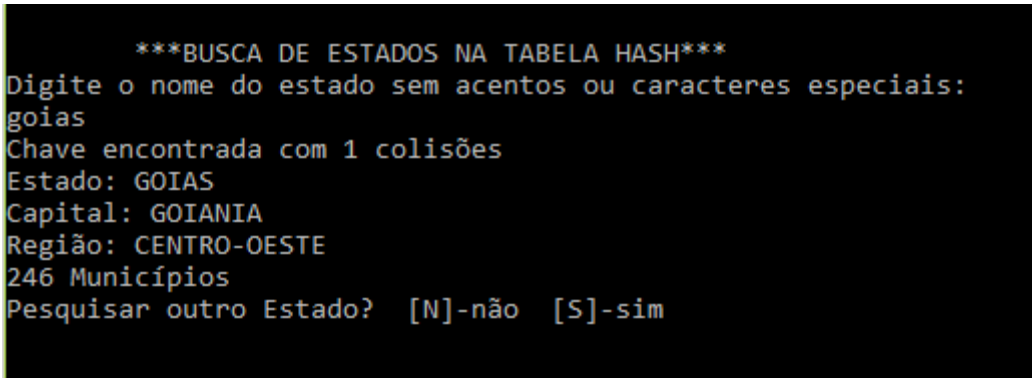

Figura 14. Teste de busca por chave GOIAS.

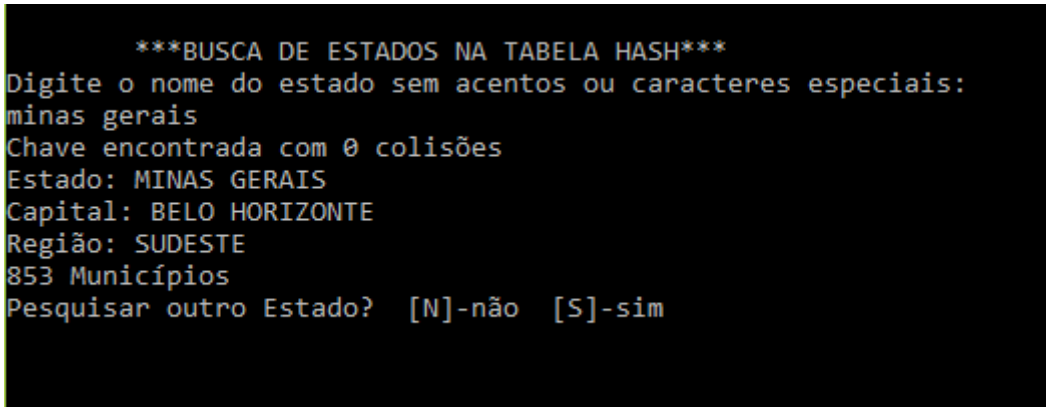

Figura 15. Teste de busca por chave MINAS GERAIS.

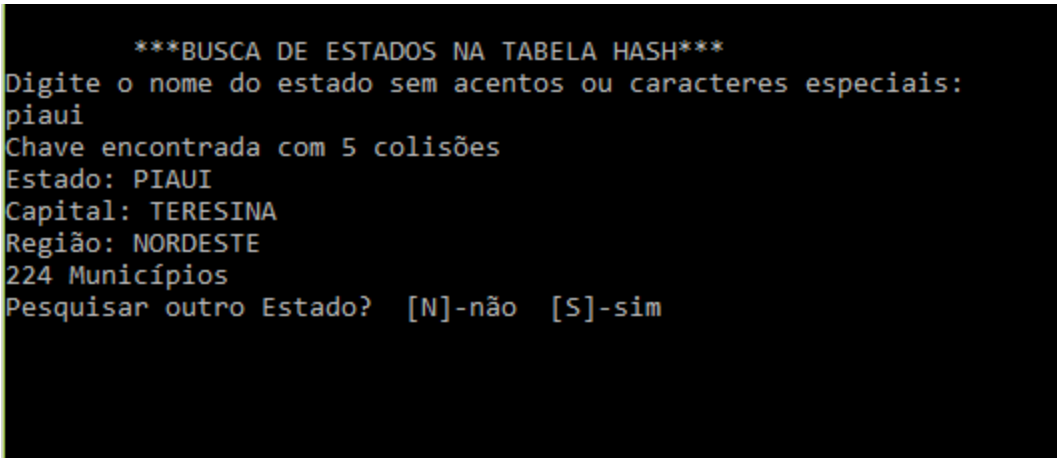

Figura 16. Teste de busca por chave PIAUI.

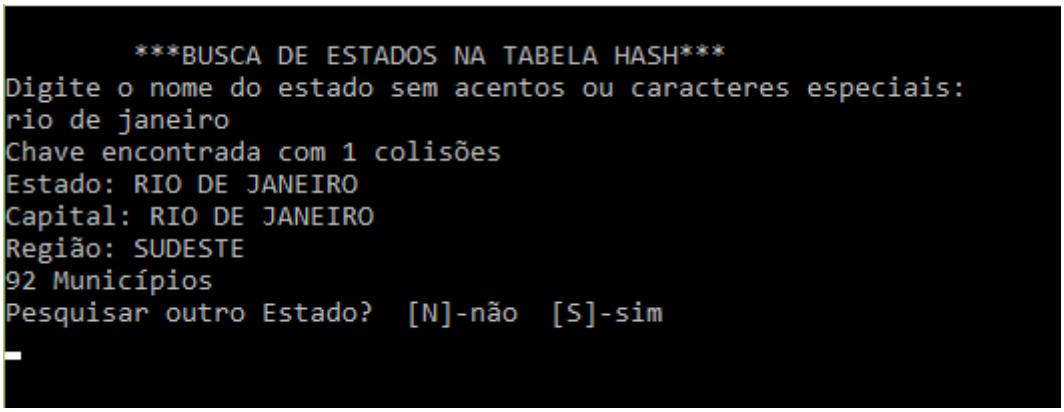

Figura 17. Teste de busca por chave RIO DE JANEIRO.

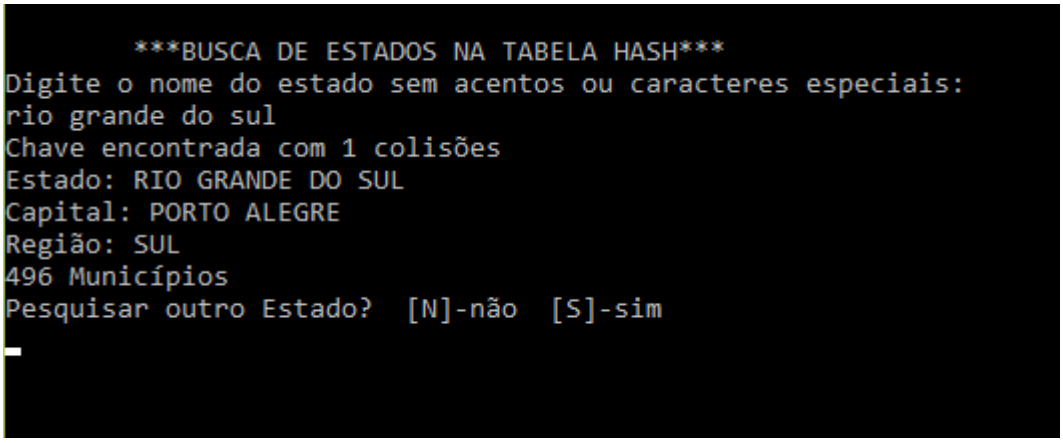

Figura 18. Teste de busca por chave RIO GRANDE DO SUL.

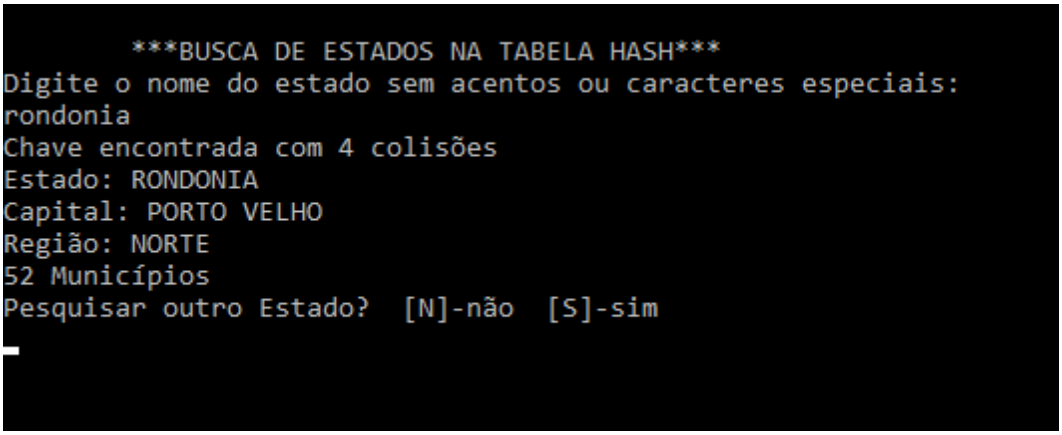

Figura 19. Teste de busca por chave RONDONIA.

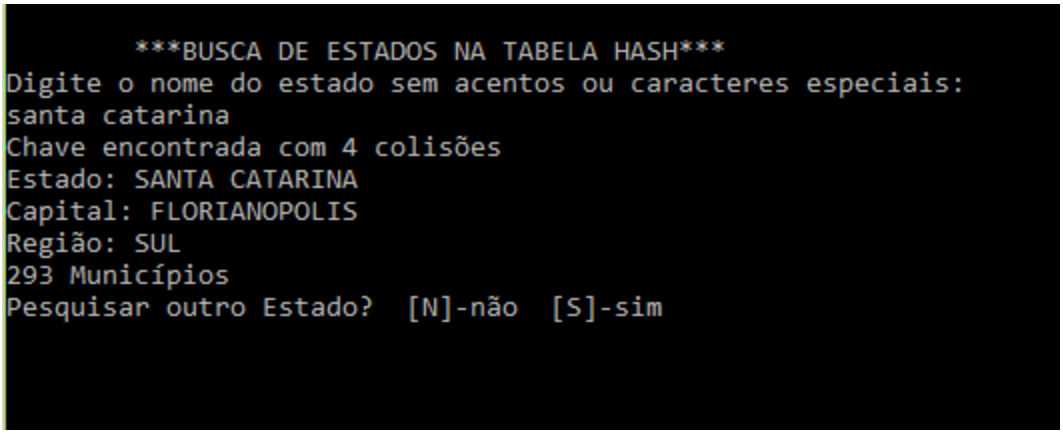

Figura 20. Teste de busca por chave SANTA CATARINA.

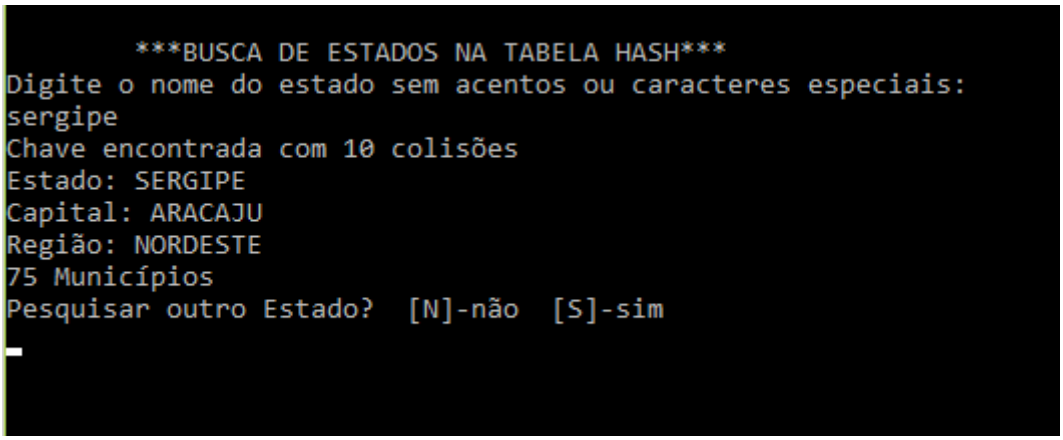

Figura 21. Teste de busca por chave SERGIPE.

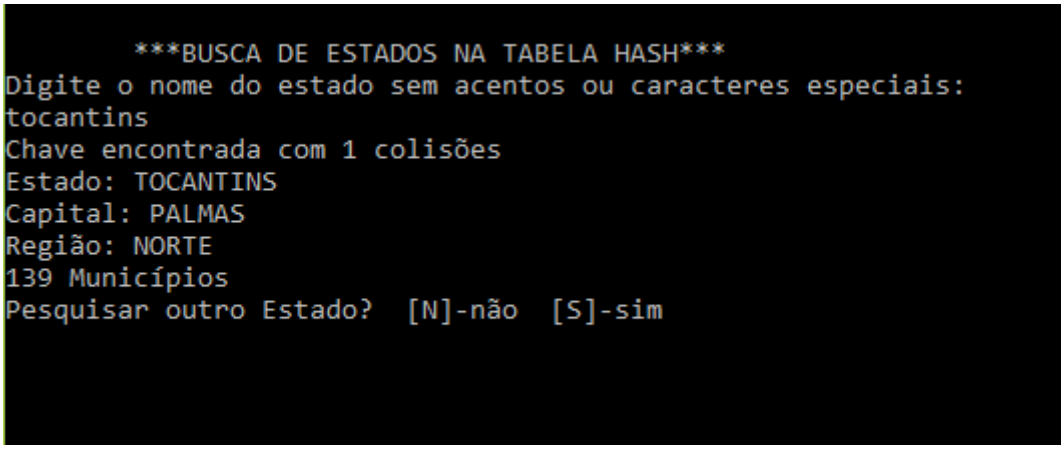

Figura 22. Teste de busca por chave TOCANTINS.

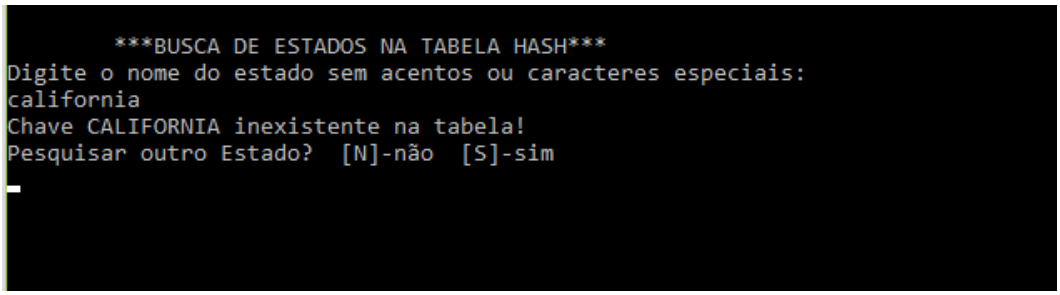

Figura 23. Teste de busca por chave inexistente.

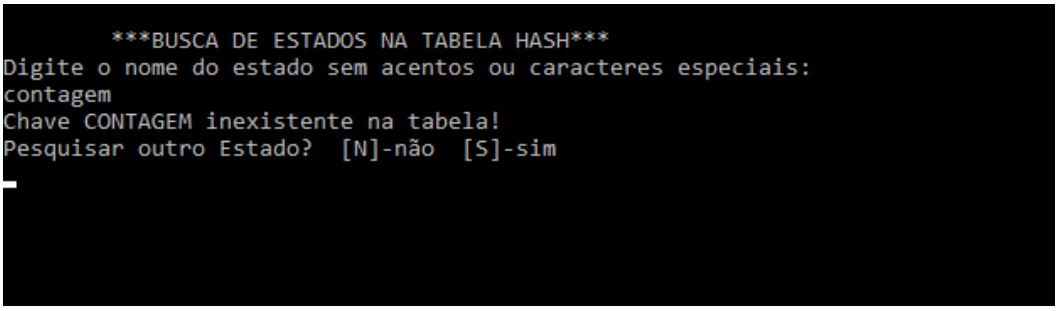

Figura 24. Teste de busca por chave inexistente.

Com o teste de visualização da Tabela Hash a seguir, é possível comparar o resultado de cada chave na função hash e a posição onde cada uma está inserida, compreendendo o número de colisões apresentado para cada busca.

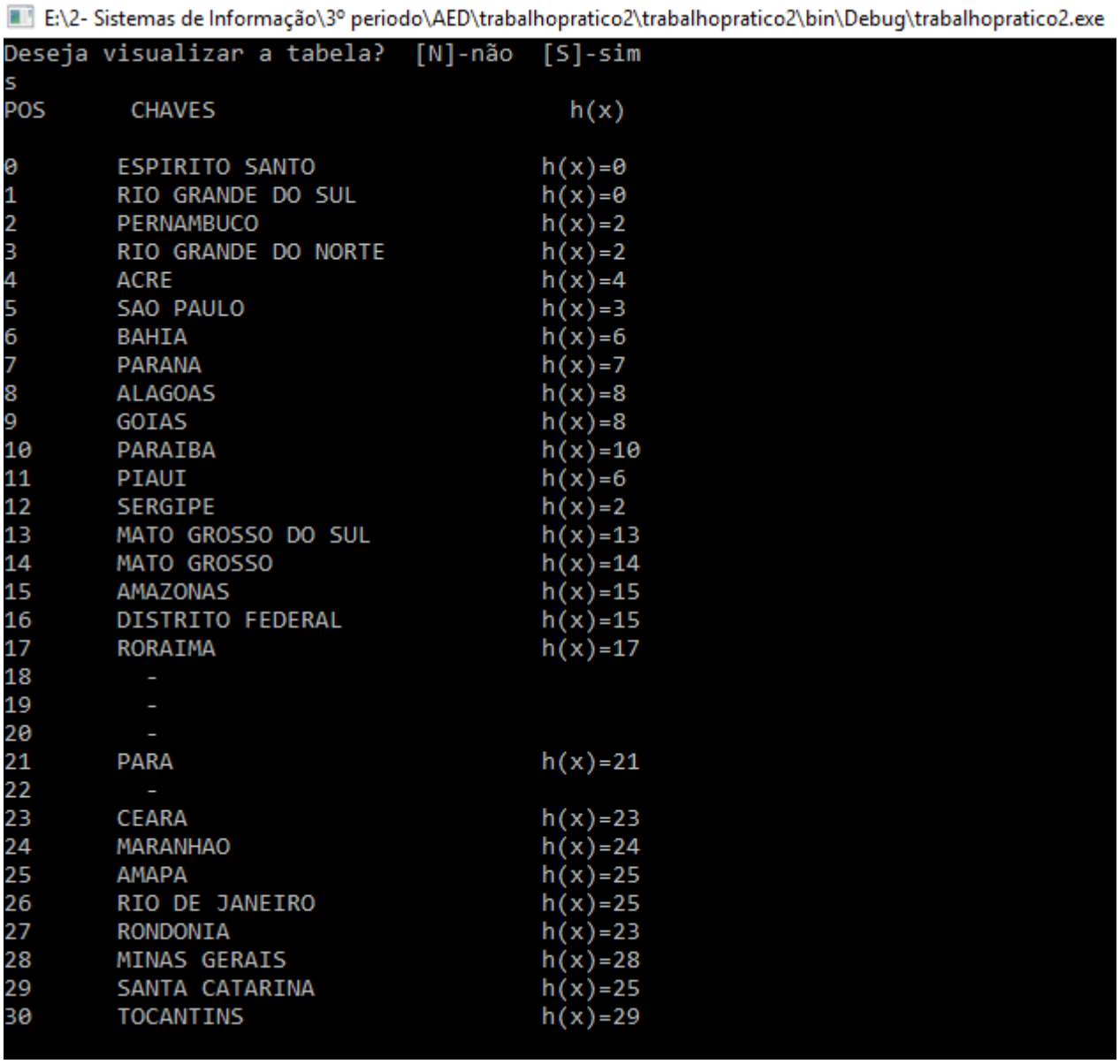

Figura 25. Teste de visualização da tabela hash, suas chaves e respectivos valores de função hash para conferência de colisões.

12. Conclusão

Com a implementação deste trabalho foi possível entendermos, principalmente por meio dos testes, como o tamanho da Tabela Hash, a elaboração da função hash e a ordem em que são inseridos os elementos podem influenciar diretamente no número de colisões e consequentemente na complexidade dos algoritmos de inserção e busca de elementos por meio de suas chaves. Também percebemos que conhecer as chaves é fundamental para construir uma Estrutura de Dados Tabela Hash. Sendo assim, cada problema que demanda um programa utilizando uma estrutura dessas para sua solução, demanda consequentemente uma Tabela Hash específica.

Referências

ESTADOSECAPITAISDOBRASIL.
Disponível em: < http://www.estadosecapitaisdobrasil.com/> Acesso em: 25 mai. 2017.

RANKBRASIL. **Estado brasileiro com maior número de municípios.** Disponível em: <http://www.rankbrasil.com.br/Recordes/Materias/06cw/Estado_Brasileiro_Com_Maior_Numero_De_Municipios> Acesso em: 25 mai. 2017.