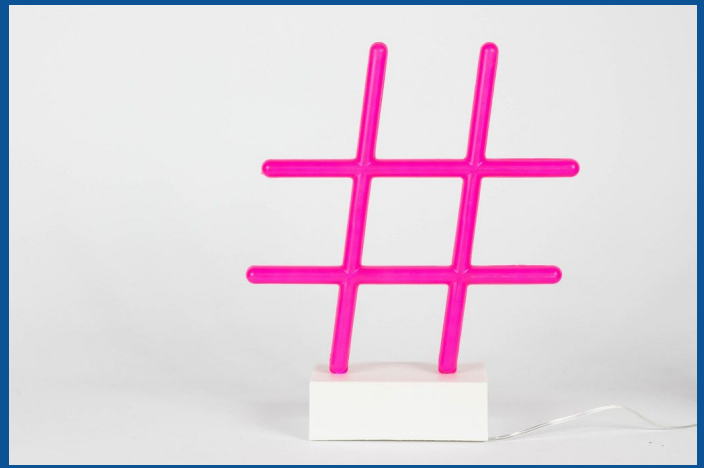
Estrutura de Dados: Tabelas Hash

Prof. Joaquim Uchôa
Profa. Juliana Greghi
Prof. Renato Ramos

- #visão geral
- #tratamento de colisão
- #tratamento de colisão por encadeamento

Visão Geral



Conceito de Tabelas Hash

Estrutura de dados que, se projetada com cuidado, permite a busca por elementos em ordem constante: $O(1)$.

Faz maior uso de memória, porém justificável parcialmente pela eficiência na busca.

Ideia Geral

Para se conseguir essa velocidade de acesso, o hash utiliza o acesso direto em posições de arranjo.

Assim, para cada valor armazenado, utiliza-se algum mecanismo (uma função de hash) que encontra a posição desse valor no arranjo.

Por Exemplo?

Suponha que queremos armazenar um conjunto de dados para consulta rápida. Esses dados possuem as seguintes chaves:

14 26 38 17 30 45

Vamos usar um arranjo de tamanho 11. Queremos acessar os elementos sem precisar percorrer o vetor. Qual função podemos usar para associar as chaves a uma posição?

Função de Espalhamento - I

Conceitualmente uma função de hash, ou escrutínio, ou função de espalhamento, é um algoritmo que mapeia dados de comprimento variável para dados de comprimento fixo.

Como vários valores podem ser mapeados em um único valor de hash, essas funções não possuem inversa, ou seja, dado um valor de hash, não é possível determinar com certeza o valor que o gerou.

Função de Espalhamento - II

Por conta dessa característica, funções de hash são a base para **armazenamento de senhas** em sistemas seguros. O sistema não armazena a senha, e sim o hash dela.

```
root:$6$eo.KHQi0w9eIv4Tc$EJryoj
yCtPoiIUzeeAg3MbAGWaY5FwuQwgzQj
UnqAj9U2pZlRAe9KomTdnjcTM2X6qkI
3zWTlnrOglyZAZMMq/:17176::::::
```

Para autenticar o usuário, o sistema verifica se o hash da senha digitada é idêntico ao hash armazenado.

Funções de Espalhamento - III

Funções de hash também são ferramentas básicas para implementação de verificações de integridade (como **checksum** de arquivos). Ao fazer um download, por exemplo, caso o checksum seja diferente do informado pelo site, o arquivo não deve ter sido transferido corretamente.

Funções de hash também são base para uma tecnologia em auge atualmente: o **blockchain**.

Funções de Espalhamento - IV

Outro uso efetivo de funções de hash são em **tabelas hash** (ou tabelas de dispersão), para encontrar a posição de uma dada chave no vetor de armazenamento.

Uma função bastante utilizada para chaves baseadas em números inteiro positivos é o operador de resto:

h(k) = k mod m

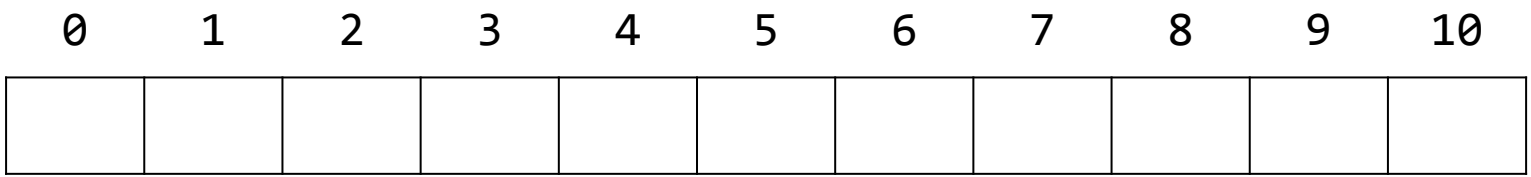
Em que k é o valor da chave e m é a capacidade da tabela.

Vamos Voltar ao Exemplo?

Para o nosso caso, já que vamos usar um arranjo de tamanho 11, podemos usar como função de hash:

h(k) = k mod 11

Com isso, teremos a posição de cada chave no vetor.



Pode Desenhar?

Aplicando-se a função de hash aos valores das chaves (14, 26, 38, 17, 30, 45), teremos a posição de cada chave no vetor:

h(14) = 3, h(26) = 4, h(38) = 5,
h(17) = 6, h(30) = 8, h(45) = 1

0	1	2	3	4	5	6	7	8	9	10
	45		14	26	38	17		30		

Calma aí, Calma aí

Construído dessa forma, perceba que a tabela hash possui mais espaço que o utilizado para alocação de dados. Isso depende não apenas na quantidade de dados, mas também da função de hash utilizada.

Uma boa função de hash pode ajudar em muito a economizar espaço na memória e a evitar colisões.

Colisão? Quem Estava Dirigindo?

Imagine que na nossa tabela hash anterior queremos agora armazenar o valor 56:

h(56) = 1

Já tem um dado nessa posição do hash!

0	1	2	3	4	5	6	7	8	9	10
	45		14	26	38	17		30		

Colisão? Quem Estava Dirigindo?

Im
qu

h(

Já

Quando ocorre de duas chaves receberem o mesmo endereço em uma tabela hash, dizemos que houve uma **colisão**.

Esse é um fenômeno não desejado e que pode ser minimizado ou até evitado com a escolha de uma boa função de hash.

Essa escolha não é trivial, entretanto...

0	1	2	3	4	5	6	7	8	9	10
	45		14	26	38	17		30		

Escolhendo Uma Função de Hash - Um Exemplo - I

O que considerar?

Um dos fatores a ser considerado no momento de escolha de uma função de hash é o tamanho do conjunto de dados de entrada.

Como fazer isso?

O primeiro passo é **conhecer o problema** a ser tratado.

Escolhendo Uma Função de Hash - Um Exemplo - II

Qual o problema a ser tratado?

Suponha que queiramos armazenar os dados de alunos matriculados nos cursos da universidade.

Cada aluno é identificado por um número de matrícula composto por 9 dígitos (ex: 201921405).

Escolhendo Uma Função de Hash - Um Exemplo - III

O que mais considerar?

Considerando 50 anos de dados, com entrada semestral de aproximadamente 3000 alunos, teríamos aproximadamente 300000 alunos cadastrados.

Caso seja utilizado um vetor normal para armazenar esses dados e ocupemos 10 KB por aluno, teremos um espaço alocado de aproximadamente 3GB de dados. O acesso aos dados não seria tão rápido, mesmo que o vetor estivesse ordenado.

Escolhendo Uma Função de Hash - Um Exemplo - iv

Possível solução?

Criar um arranjo em que a posição de cada aluno é dada por sua matrícula. Uma tabela hash.

Com isso, o acesso aos dados de um aluno seria praticamente instantâneo.

Escolhendo Uma Função de Hash - Um Exemplo - V

Qual o problema?

Certo,... para acesso constante, o número de matrícula pode ser usado como índice em um arranjo (vetor).

Como a matrícula é composta por 9 dígitos, os valores variam de 0 a 999 999 999.

O arranjo criado deveria ter dimensão
1 000 000 000 !!!!!

Escolhendo Uma Função de Hash - Um Exemplo - VI

Qual o problema?

Caso seja utilizado um registro com cerca de 10 KB por aluno, a quantidade de memória dispendida seria de

10 * 1 000 000 000 = 10 000 000 000 KB ≃ 10 TB

Escolhendo Uma Função de Hash - Um Exemplo - VII

Existem alternativas?

Percebam que o arranjo teria muitas posições vazias.

Caso encontremos uma função que possa associar uma matrícula a uma única posição do arranjo, diminuindo o espaço vazio, conseguiríamos então acesso constante aos dados.

Conceito Formal de Tabela Hash

O objetivo de uma tabela hash é possibilitar armazenar os dados e identificar, através de uma chave de busca, a posição na tabela ocupada por determinado dado.

Tempo de busca reduzido de $O(n)$, busca sequencial, ou $O(\lg n)$, busca binária, para $O(1)$.

Usa-se função de espalhamento (*hash function*) para transformar uma chave K em uma posição na tabela.

Função de Espalhamento - 1/2

A função de espalhamento (*hash function*) associa uma chave de busca a um índice na tabela.

Não é tarefa fácil encontrar a função de hash mais adequada para um dado problema.

Uma boa função de hash satisfaz a suposição de que é possível que cada chave seja atribuída a uma posição diferente da tabela independentemente da posição em que as demais chaves foram atribuídas.

Função de espalhamento - 2/2

Diante da dificuldade de encontrar uma função de hash mais adequada, geralmente se utilizam funções de hash tradicionais, como a de módulo, já apresentada anteriormente:

h(k) = k mod m

onde k é o valor da chave e m é a quantidade de posições da tabela. Na prática, costuma-se adotar um valor primo como divisor, para minimizar colisões.

Implementação Usual de Tabela Hash

- Tabelas hash são comumente implementadas em arranjos, com tamanho definido de forma estática.
- Valores retornados pela função hash correspondem aos índices do arranjo.
- Não se tem conhecimento prévio da dimensão necessária.
- Mesmo com estratégias para tratamento de colisões, estrutura pode ser descaracterizada se a taxa de ocupação for muito alta.

Tabela Hash dinâmica

Algumas abordagens permitem o redimensionamento da tabela quando a taxa de ocupação ultrapassa determinada limite. O redimensionamento é um processo caro, mas mantém o desempenho médio da tabela.

Após o redimensionamento, todos os dados precisam ser realocados (copiar dados de um arranjo para outro).

Alguns autores chamam essa técnica de tabela hash dinâmica (mesmo com dados armazenados em arranjos).

Tratamento de Colisão

E se um dado, após a aplicação da função hash à sua chave, tiver que ocupar uma posição já ocupada?

E se um dado, após a aplicação da função hash à sua chave, tiver que ocupar uma posição já ocupada?

Será necessário tratar a ocorrência de colisão, o que pode ser feito de duas maneiras.

E se um dado, após a aplicação da função hash à sua chave, tiver que ocupar uma posição já ocupada?

Será necessário tratar a ocorrência de colisão, o que pode ser feito de duas maneiras:

- Encadeamento
- Endereçamento aberto

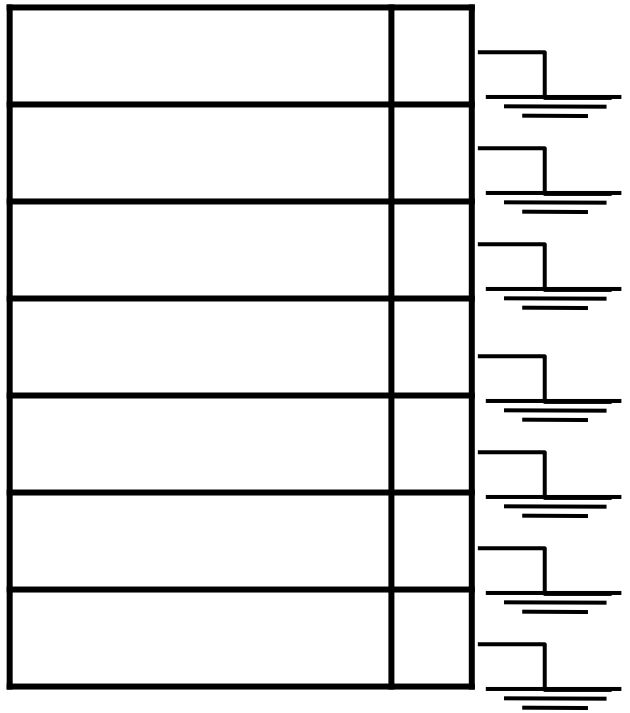
Tratamento de Colisão - Encadeamento 1/2

Se a função hash mapeia a chave de busca para uma posição já ocupada, o elemento armazenado nesta posição utiliza uma lista encadeada, em que são armazenados os elementos que sofreram colisão para aquela chave.

Tratamento de Colisão - Encadeamento 2/2

Cada posição na tabela pode ser entendido como um elemento que aponta para uma lista encadeada.

Alternativamente, pode-se implementar a tabela hash como um vetor de listas encadeadas.



Exemplo - 1/9

Capacidade da tabela: 11

Chaves dos dados a serem inseridos:

34 46 101 85

55 13 96

Função: h(k) = k mod m

Exemplo 2/9

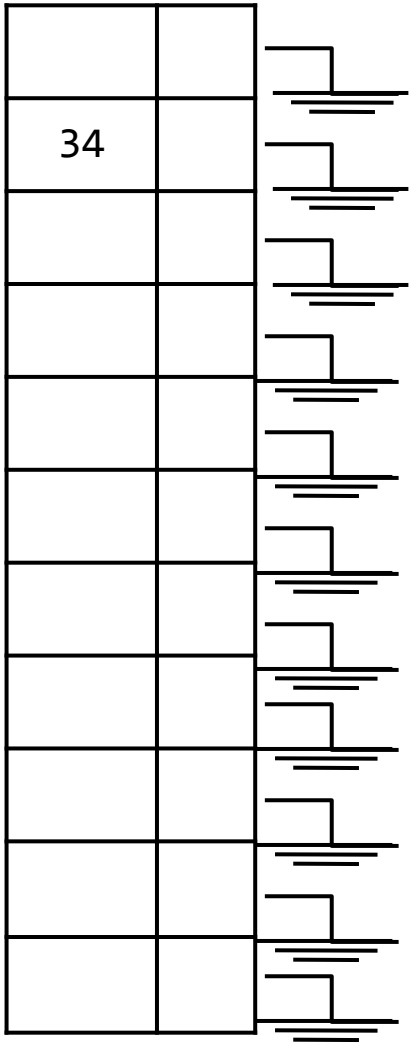
Capacidade da tabela: 11

Chaves dos dados a serem inseridos:

~~34~~ 46 101 85

55 13 96

Função: h(k) = k mod m



Exemplo 3/9

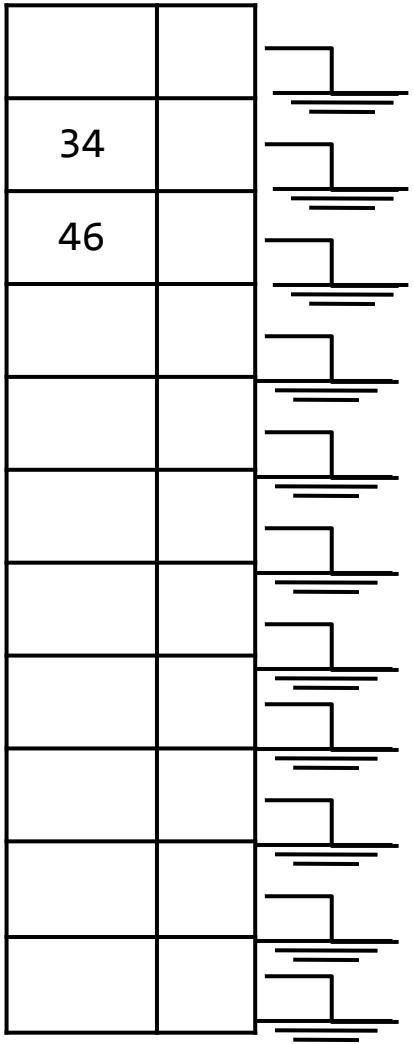
Capacidade da tabela: 11

Chaves dos dados a serem inseridos:

~~34~~ ~~46~~ 101 85

55 13 96

Função: h(k) = k mod m



Exemplo 4/9

Capacidade da tabela: 11

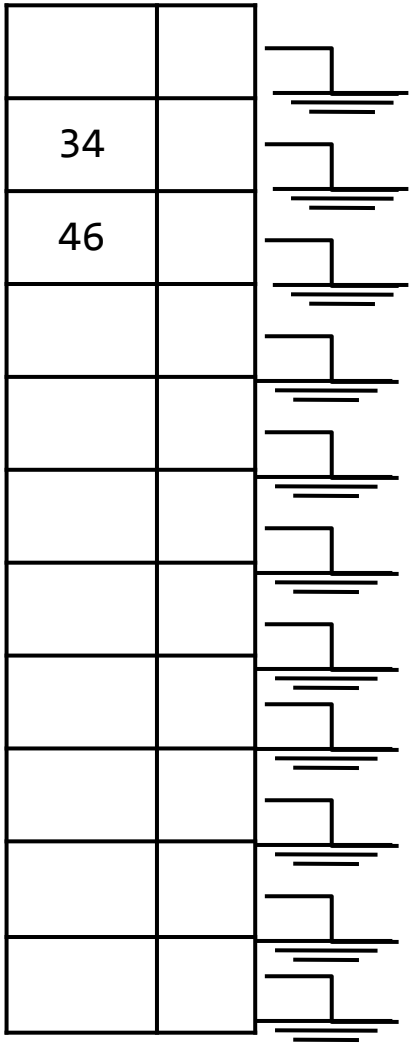
Chaves dos dados a serem inseridos:

~~34~~ ~~46~~ 101 85

55 13 96

Função: h(*k*) = *k* mod *m*

$h(101) = 101 \bmod 11 = 2$ ➡ **Colisão!**



Exemplo 5/9

Capacidade da tabela: 11

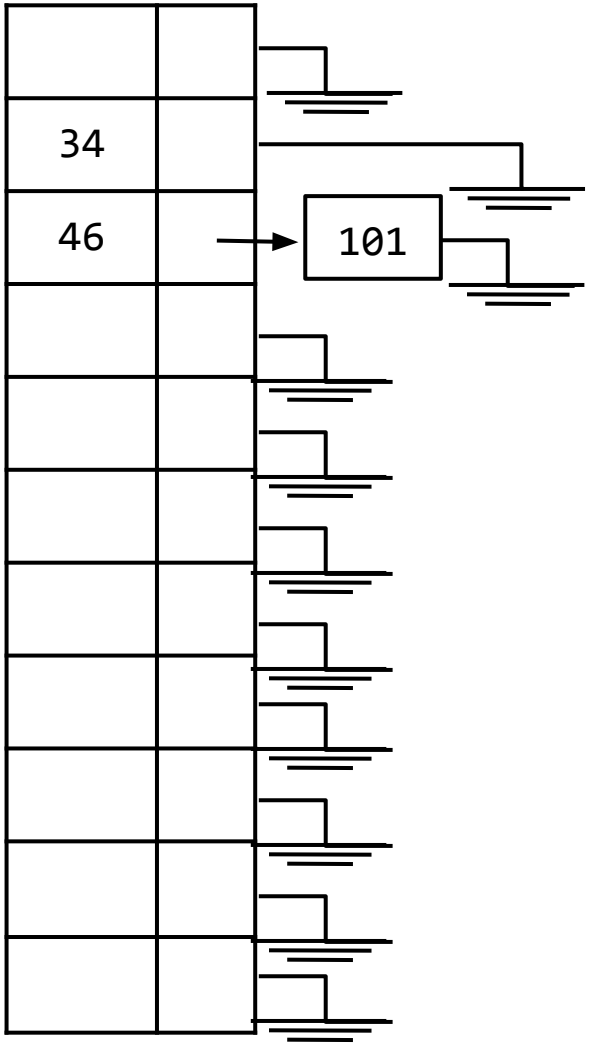
Chaves dos dados a serem inseridos:

~~34~~ ~~46~~ ~~101~~ 85

55 13 96

Função: h(*k*) = *k* mod *m*

$h(101) = 101 \bmod 11 = 2$ ➡ **Colisão!**



Exemplo 6/9

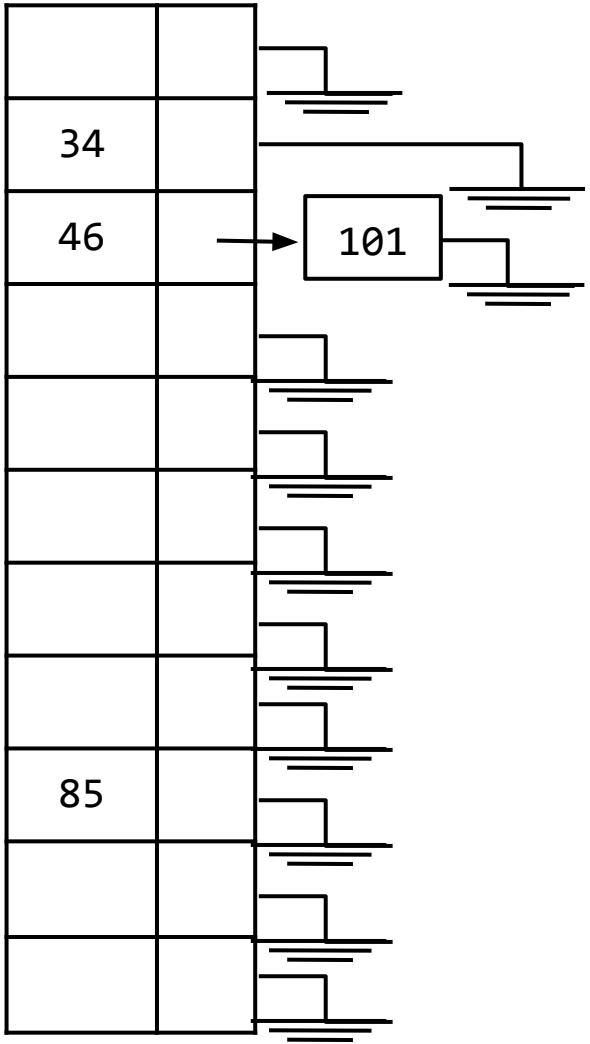
Capacidade da tabela: 11

Chaves dos dados a serem inseridos:

~~34~~ ~~46~~ ~~101~~ ~~85~~

55 13 96

Função: h(k) = k mod m



Exemplo 7/9

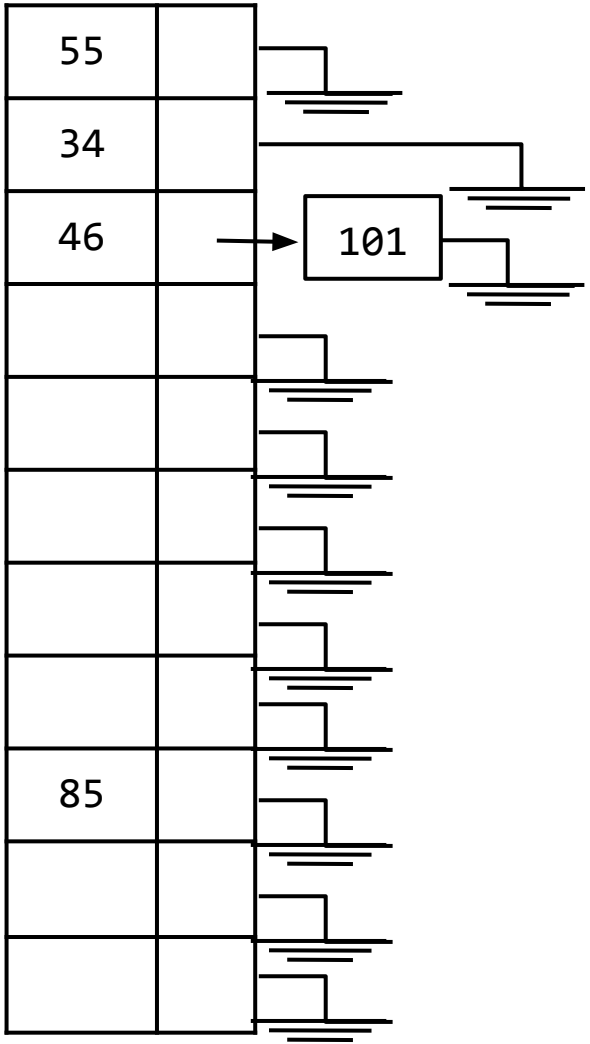
Capacidade da tabela: 11

Chaves dos dados a serem inseridos:

~~34~~ ~~46~~ ~~101~~ ~~85~~

~~55~~ 13 96

Função: h(k) = k mod m



Exemplo 8/9

Capacidade da tabela: 11

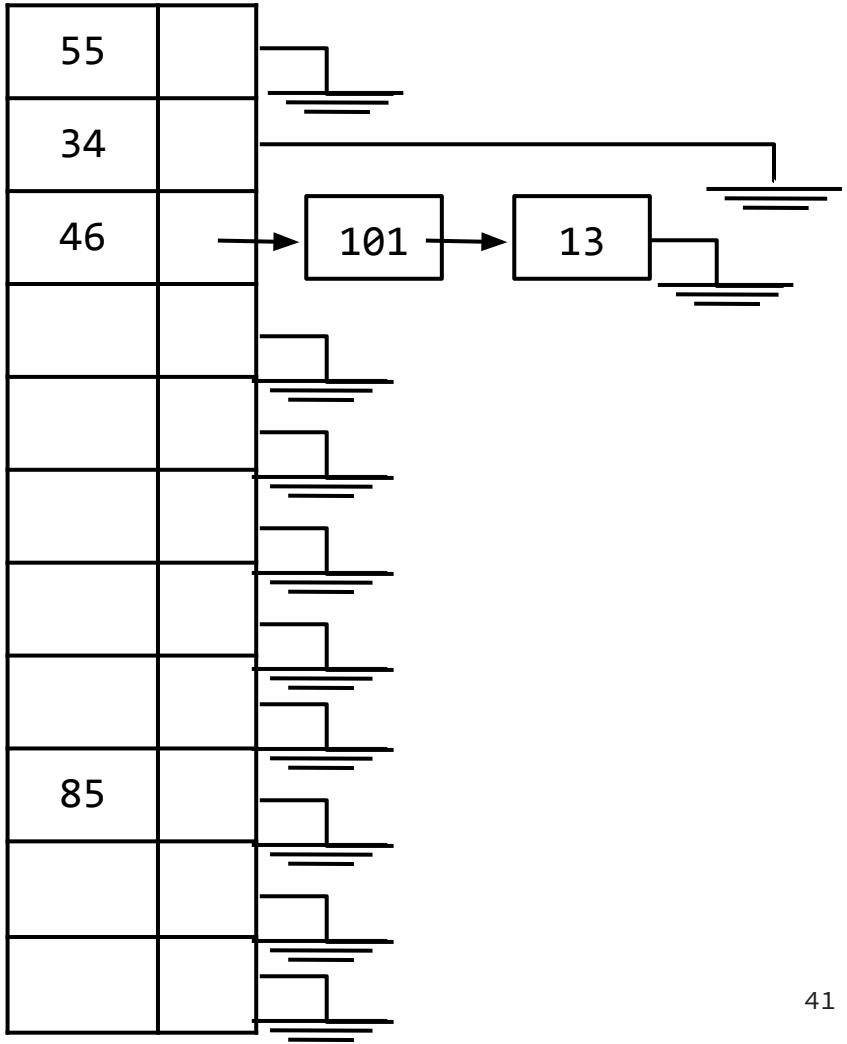
Chaves dos dados a serem inseridos:

~~34~~ ~~46~~ ~~101~~ ~~85~~

~~55~~ ~~13~~ 96

Função: h(*k*) = *k* mod *m*

h(101) = *13* mod *11* = 2 ➡ **Colisão!**



Exemplo 9/9

Capacidade da tabela: 11

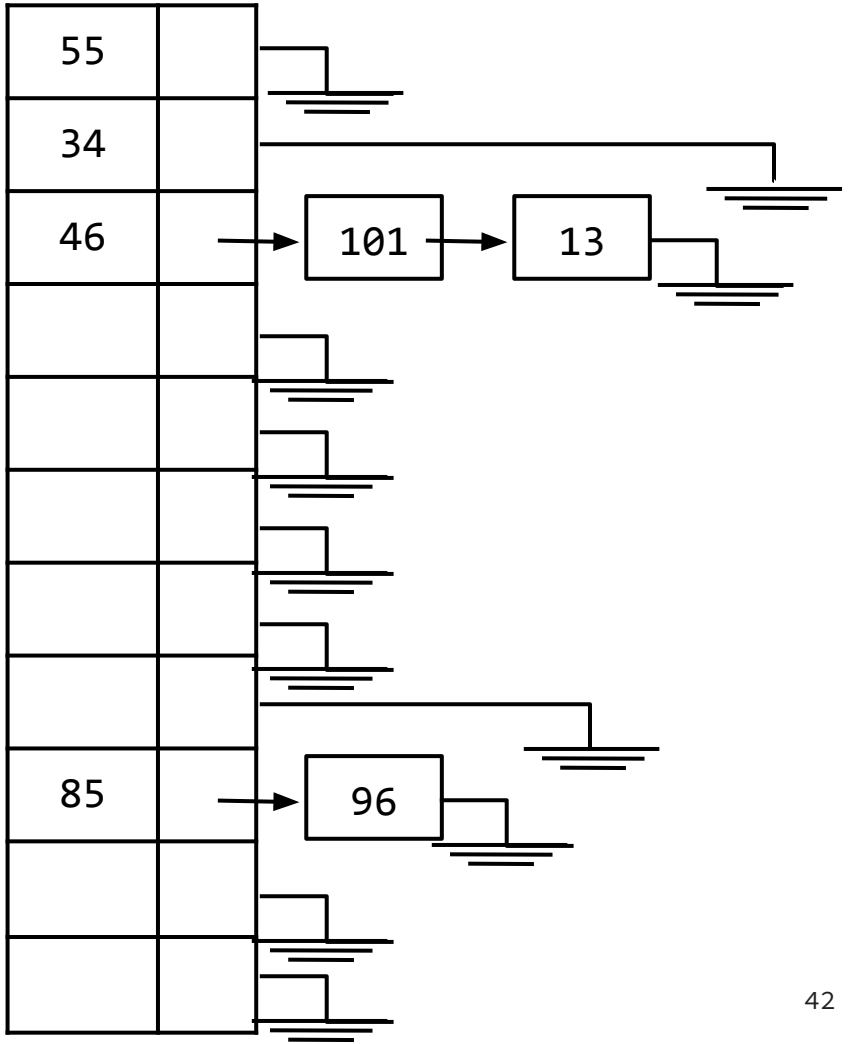
Chaves dos dados a serem inseridos:

~~34~~ ~~46~~ ~~101~~ ~~85~~

~~55~~ ~~13~~ ~~96~~

Função: h(*k*) = *k* mod *m*

h(101) = *96* mod *11* = 8 ➡ **Colisão!**



Observação Importante

Nos exemplos apresentados, a inserção dos elementos que geraram colisão está sendo feita no final da lista.

Entretanto, essa inclusão também poderia ser realizada no início da lista, sendo apenas uma decisão de implementação.

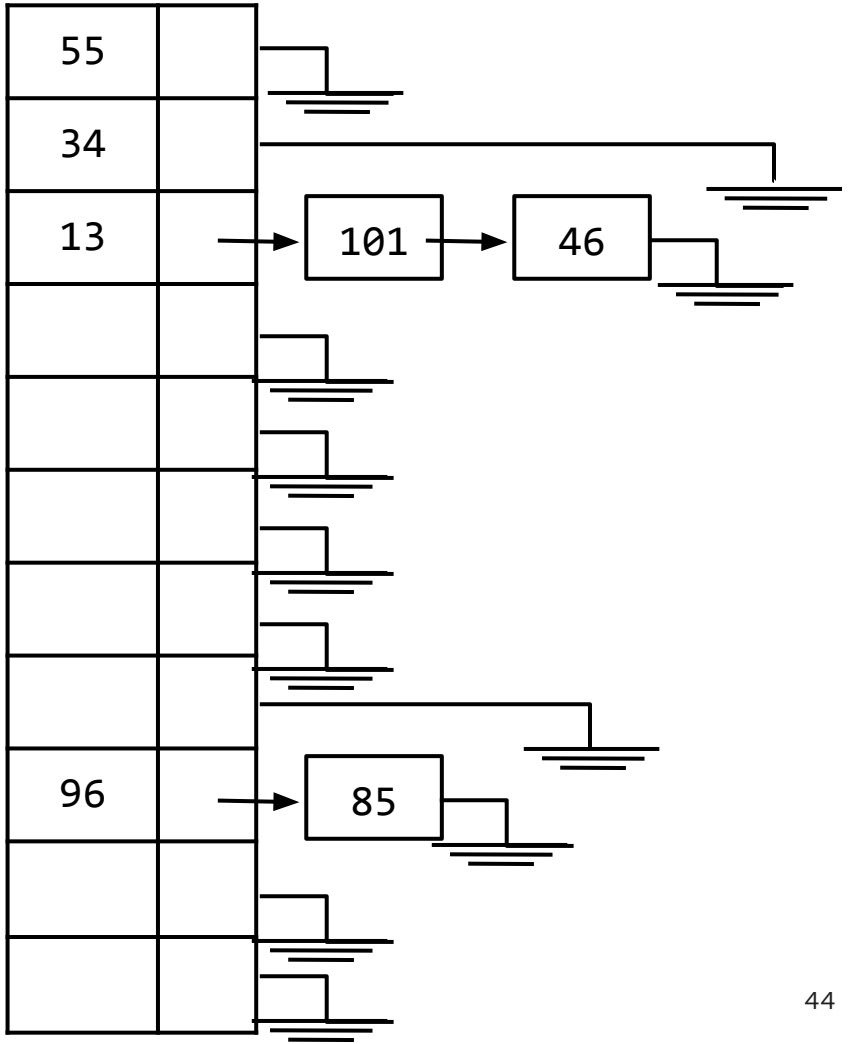
Uma das vantagens da inserção no início é não precisar implementar um ponteiro para o final da lista.

Exemplo

Considerando capacidade da tabela igual a 11, usando $h(k) = k \bmod m$.

Sejam as seguintes chaves: 34, 46, 101, 85, 55, 13, 96.

Caso a colisão seja inserida no início da lista de colisão, então tem-se a situação ao lado.



Tratamento de Colisão - Endereçamento Aberto - 1/2

Se a função hash mapeia a chave de busca para uma posição já ocupada, procura-se a próxima posição livre para armazenar o novo elemento.

Uma tabela nunca estará completamente preenchida - estudos mostram que com a ocupação acima de 75% o número de colisões é muito grande, o que descaracterizaria a ideia principal da estrutura.

Tratamento de Colisão - Endereçamento Aberto - 2/2

No caso de uma busca, se uma chave *k* for mapeada para uma determinada posição, e o elemento buscado não for o desejado, procura-se nas posições subsequentes até que seja encontrado ou que uma posição vazia seja encontrada.

Exemplo 1/5

Capacidade da tabela: 7

Chaves dos dados a serem inseridos:

~~34~~ 46 85 55 13

$$h(34) = 34 \bmod 7$$

	0
	1
	2
	3
	4
	5
34	6

Exemplo 2/5

Capacidade da tabela: 7

Chaves dos dados a serem inseridos:

~~34~~ ~~46~~ 85 55 13

$$h(46) = 46 \bmod 7$$

	0
	1
	2
	3
46	4
	5
34	6

Exemplo 3/5

Capacidade da tabela: 7

Chaves dos dados a serem inseridos:

~~34~~ ~~46~~ ~~85~~ 55 13

$$h(85) = 85 \bmod 7$$

	0
85	1
	2
	3
46	4
	5
34	6

Exemplo 4/5

Capacidade da tabela: 7

Chaves dos dados a serem inseridos:

~~34~~ ~~46~~ ~~85~~ 55 13

$$h(55) = 55 \bmod 7 = 6$$ **Colisão !**

	0
85	1
	2
	3
46	4
	5
34	6

Exemplo 4/5

Capacidade da tabela: 7

Chaves dos dados a serem inseridos:

~~34~~ ~~46~~ ~~85~~ 55 13

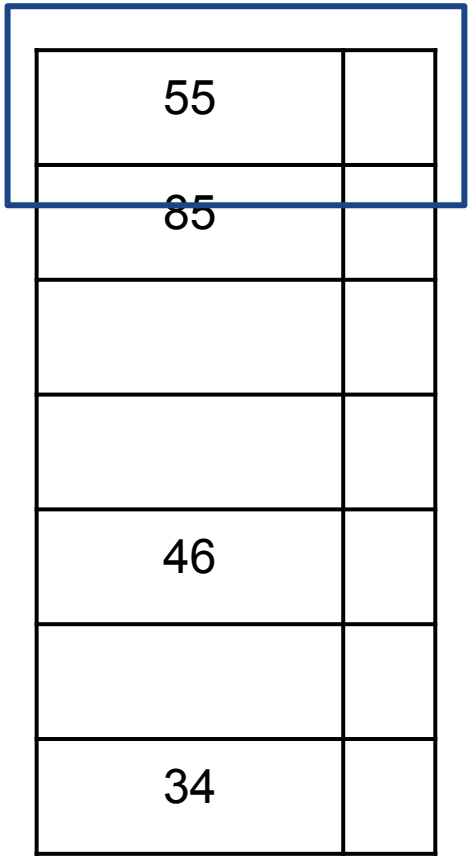
Próxima posição vazia????

$h(55) = 55 \bmod 7 = 6$ **Colisão !**

	0
85	1
	2
	3
46	4
	5
34	6

Exemplo 4/5

O tratamento de colisão por endereçamento aberto utiliza uma lista circular para a busca e inserção de novos elementos.



Exemplo 4/5

Capacidade da tabela: 7

Chaves dos dados a serem inseridos:

~~34~~ ~~46~~ ~~85~~ ~~55~~ 13

$$h(55) = 55 \bmod 7 = 6$$

55	0
85	1
	2
	3
46	4
	5
34	6

Exemplo 5/5

Capacidade da tabela: 7

Chaves dos dados a serem inseridos:

~~34~~ ~~46~~ ~~85~~ ~~55~~ 13

$$h(13) = 13 \bmod 7 = 6$$ **Colisão !**

55	0
85	1
	2
	3
46	4
	5
34	6

Exemplo 5/5

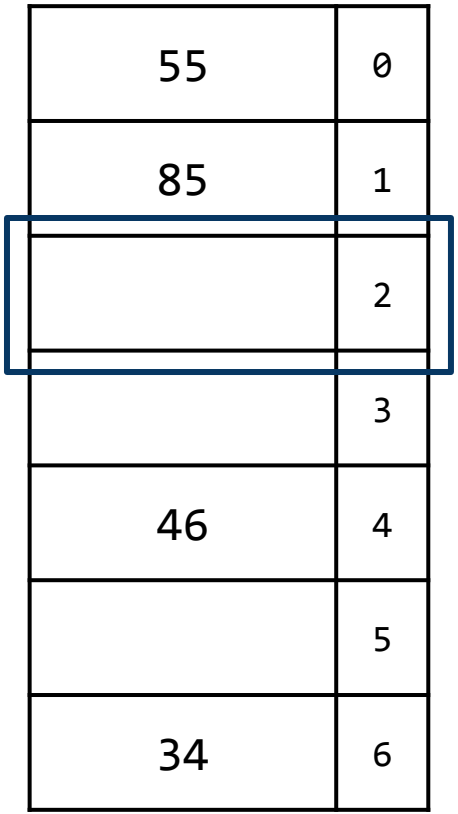
Capacidade da tabela: 7

Chaves dos dados a serem inseridos:

~~34~~ ~~46~~ ~~85~~ ~~55~~ 13

Próxima posição vazia????

$h(13) = 13 \bmod 7 = 6$ **Colisão !**



Exemplo 5/5

Capacidade da tabela: 7

Chaves dos dados a serem inseridos:

~~34~~ ~~46~~ ~~85~~ ~~55~~ ~~13~~

$$h(13) = 13 \bmod 7 = 6$$

55	0
85	1
13	2
	3
46	4
	5
34	6

Remoção de Elementos

- Com tratamento de colisão por encadeamento a remoção de um elemento é simples, uma vez que é uma remoção em lista.
- Com endereçamento aberto, o esforço é relativamente maior.

Remoção de Elemento - Endereçamento Aberto

Primeira estratégia:

- Deslocamento dos dados para a posição desocupada
  - Remover o 85

55	
85	
13	
17	
46	
34	

Remoção de Elemento - Endereçamento Aberto

É necessário verificar, enquanto não se retorna à posição inicial ou chega-se em uma posição vazia, se o elemento não deve ser reposicionado.

17 e 46 não mudam de posição, por exemplo.

55	
13	
17	
46	
34	

Remoção de Elemento - Endereçamento Aberto

Segunda estratégia:

- Marcar a posição desocupada como disponível para uso, preenchendo a posição, por exemplo, com o valor -1, caso os dados sejam todos positivos.

55	
-1	
13	
17	
46	
34	

Remoção de Elemento - Endereçamento Aberto

Caso haja um grande número de remoções usando a segunda estratégia, a busca tornar-se-á ineficiente.

Nesse caso, uma solução é a re-inserção dos elementos não removidos na tabela de modo a tornar as posições não ocupadas livres, mas mantendo a eficiência da estrutura.

Implementação de Tabela Hash com Encadeamento

Tabela Hash com Encadeamento - Visão Geral

A implementação de tabela hash com encadeamento utiliza geralmente um vetor de listas. Em alguns casos, a lista é implícita (ou seja um vetor de ponteiros de nós encadeados).

Os elementos usualmente constituem um par (chave,valor). Dada uma chave, a intenção é recuperar o valor armazenado anteriormente associado a esta chave. Cada elemento é um nó em uma das listas no vetor de listas.

Tabela Hash com Encadeamento - Criação/Destruição

criarHash(cap):

```
capacidade ← cap;
tabela ← alocaVetorDeListas(capacidade);
```

destruirHash():

```
desalocaVetorDeListas();
```

Tabela Hash com Encadeamento - Inserção

insereElemento(valor,chave):

```
posicao ← calculaHash(chave);
lista ← tabela[posicao];
se ( lista.busca(chave) = NAOENCONTRADO ) {
   // cria um elemento e o insere no final da lista
   lista.insereNoFim(valor,chave);
} senão {
   geraErro("ITEM JÁ ESTÁ NA TABELA!");
}
```

Tabela Hash com Encadeamento - Inserção

insereElemento(valor,chave):

```
posicao ← calculaHash(chave);
lista ← tabela[posicao];
se ( lista.busca(chave) = NAOENCONTRADO ) {
   // cria no final da lista
   lista.in
} senão {
   geraErro ;
}
```

Objeto devolvido é uma referência ou ponteiro, não uma cópia.

Tabela Hash com Encadeamento - Inserção

insereElemento(valor,chave):

```
posicao ← calculaHash(chave);
lista ← tabela[posicao];
se ( lista.busca(chave) =
   // cria um elemento e o
   lista.insereNoFim(valor
} senão {
   geraErro("ITEM JÁ ESTÁ NA TABELA!");
}
```

Vai permitir dois elementos com mesma chave? Se sim, quando for buscar, o que acontece nesse caso?

Tabela Hash com Encadeamento - Inserção

insereElemento(valor,chave):

```
posicao ←
lista ←
se ( list                                    {
   // cria um                    e o insere no final da lista
   lista.insereNoFim(valor,chave);
} senão {
   geraErro("ITEM JÁ ESTÁ NA TABELA!");
}
```

Também poderia ser utilizado `insereNoInicio()`, caso seja mais interessante para a implementação.

Tabela Hash com Encadeamento - Recuperação

recuperaValor(chave):

```
posicao ← calculaHash(chave);
elemento ← tabela[posicao].busca(chave);
se ( elemento = NAOENCONTRADO ) {
   geraErro("ITEM NÃO ENCONTRADO!");
} senão {
   efetuaAcao(elemento.valor);
}
```

Tabela Hash com Encadeamento - Alteração

alteraElemento(chave,novoValor):

```
posicao ← calculaHash(chave);
elemento ← tabela[posicao].busca(chave);
se ( elemento = NAOENCONTRADO ) {
   geraErro("ITEM NÃO ENCONTRADO!");
} senão {
   elemento.valor ← novoValor;
}
```

Tabela Hash com Encadeamento - Alteração

```
alteraElemento(chave,novoValor);
posicao ← calculaHash(cha
elemento ← tabela[posicao
se ( elemento = NAOENCONTR
   geraErro("ITEM NÃO ENCONTRADO! ");
} senão {
   elemento.valor ← novoValor;
}
```

Caso elemento não exista, não é melhor criá-lo que gerar o erro? Existência anterior é necessária para alteração?

Tabela Hash com Encadeamento - Remoção

removeElemento(chave):

```
posicao ← calculaHash(chave);
elemento ← tabela[posicao].busca(chave);
se ( elemento = NAOENCONTRADO ) {
   geraErro("ITEM NÃO ENCONTRADO!");
} senão {
   tabela[posicao].apagar(elemento);
}
```

Tabela Hash com Encadeamento - Redimensionamento

redimensiona(novaCapacidade):

```
tabelaAux ← alocaVetorDeListas(novaCapacidade);
para (cada elemento em tabela) {
   insereElementoNaTabela(elemento,tabelaAux);
}

capacidade ← novaCapacidade;
apaga(tabela);
// redireciona a tabela para a com nova capacidade
tabela ← tabelaAux;
```

Implementação de Tabela Hash com Endereçamento aberto

Tabela Hash com Endereçamento Aberto- Visão Geral

A implementação de tabela hash com endereçamento aberto utiliza geralmente um vetor de dados. Isso é válido principalmente se os valores armazenados são inteiros positivos, facilitando a marcação de dados inválidos ou removidos.

Os elementos geralmente constituem um par (chave,valor). Dada uma chave, a intenção é recuperar o valor armazenado anteriormente associado a esta chave.

Tabela Hash com End. Aberto - Criação/Destruição

criarHash(cap):

```
capacidade ← cap;
tamanho ← 0;
vetorDeDados ← alocaVetorDeDados(capacidade);
para (todo elemento no vetorDeDados) {
    elemento ← INVALIDO;
}
```

destruirHash():

```
desalocaVetorDeDados();
```

Tabela Hash com End. Aberto - Buscar Chave

buscaChave(chave):

```
posicao ← calculaHash(chave);
posFinal ← posicao
faça {
  umDado ← vetorDeDados[posicao];
  se (umDado = INVALIDO) retorna POSINVALIDA;
  se (umDado.chave = chave) retorna posicao;
  posicao ← (posicao + 1) % capacidade;
} enquanto (posicao ≠ posFinal);
retorna POSINVALIDA; // percorreu o vetor e não encontrou
```

Tabela Hash com End. Aberto - Inserção I

insereElemento(valor,chave):

```
se (tamanho = capacidade)
  geraErro("TABELA HASH CHEIA!");
se ( busca(chave) ≠ POSINVALIDA )
  geraErro("ITEM JÁ ESTÁ NA TABELA!");
posicao ← calculaHash(chave);
...
```

Tabela Hash com End. Aberto - Inserção II

```
enquanto ((vetorDeDados[posicao] ≠ INVALIDO) e
  (vetorDeDados[pos] ≠ REMOVIDO))
    posicao ← (posicao + 1) % capacidade;

vetorDeDados[posicao].chave ← chave
vetorDeDados[posicao].valor ← valor;
tamanho++;
```

Tabela Hash com End. Aberto - Remove Elemento

removeElemento(chave):

```
se (tamanho = 0)
  geraErro("REMOÇÃO EM TABELA HASH VAZIA!");
posicao ← buscaChave(chave);
se (posicao = POSINVALIDA)
  geraErro("CHAVE NÃO ENCONTRADA PARA REMOÇÃO");
vetorDeDados[posicao] ← REMOVIDO;
tamanho--;
```

Tabela Hash com End. Aberto - Consultar Dado

consultaDado(chave):

```
posicao ← buscaChave(chave);
se (posicao = POSINVALIDA)
  geraErro("ELEMENTO NÃO ENCONTRADO");
dado ← vetorDeDados[posicao].valor;
efetuaAcao(dado); //possivelmente retorno
```

Sobre o Material

Sobre este material

Material produzido coletivamente, principalmente pelos seguintes professores do DCC/UFLA:

- Joaquim Quinteiro Uchôa
- Juliana Galvani Greghi
- Renato Ramos da Silva

Inclui contribuições de outros professores do setor de Fundamentos de Programação do DCC/UFLA.