

Estruturas de Dados

Tabela Hash

Prof. André Luiz Moura
<andreluiz@inf.ufg.br>



INSTITUTO DE
INFORMÁTICA
UFG

Tabela Hash: Problema

- Princípio de funcionamento dos métodos de busca
 - ▣ Procurar a informação desejada com base na comparação de suas chaves, isto é com base em algum valor que a compõe
- Problema
 - ▣ Algoritmos eficientes necessitam que os elementos estejam armazenados de forma ordenada
 - ▣ Custo ordenação melhor caso é **$O(N \log N)$**
 - ▣ Custo da busca melhor caso é **$O(\log N)$**

Tabela Hash: Problema

- Custo da comparação de chaves é alto
- O que seria uma operação de **busca ideal**?
 - ▣ Seria aquela que permitisse o acesso direto ao elemento procurado, sem nenhuma etapa de comparação de chaves
 - ▣ Nesse caso, teríamos um custo **$O(1)$**
 - Tempo sempre constante de acesso

Tabela Hash: Problema

- Uma saída é usar **arrays**
 - ▣ São estruturas que utilizam índices para armazenar informações
 - ▣ Permite acessar uma determinada posição com custo **$O(1)$**
- Problema
 - ▣ Arrays não possuem nenhum mecanismo que permita calcular a posição onde uma informação está armazenada
 - ▣ A operação de busca não é **$O(1)$**

Tabela Hash: Problema

- Precisamos do tempo de acesso do array juntamente com a capacidade de busca um elemento em tempo constante
- Solução: usar uma **tabela hash**

Tabela Hash

- Também conhecidas como tabelas de indexação ou de espalhamento
 - ▣ É uma generalização da idéia de array.
- Ideia central
 - ▣ Utilizar uma função, chamada de **função de hashing**, para espalhar os elementos que queremos armazenar na tabela.
 - ▣ Esse espalhamento faz com que os elementos fiquem dispersos de forma não ordenada dentro do array que define a tabela

Tabela Hash

Exemplo

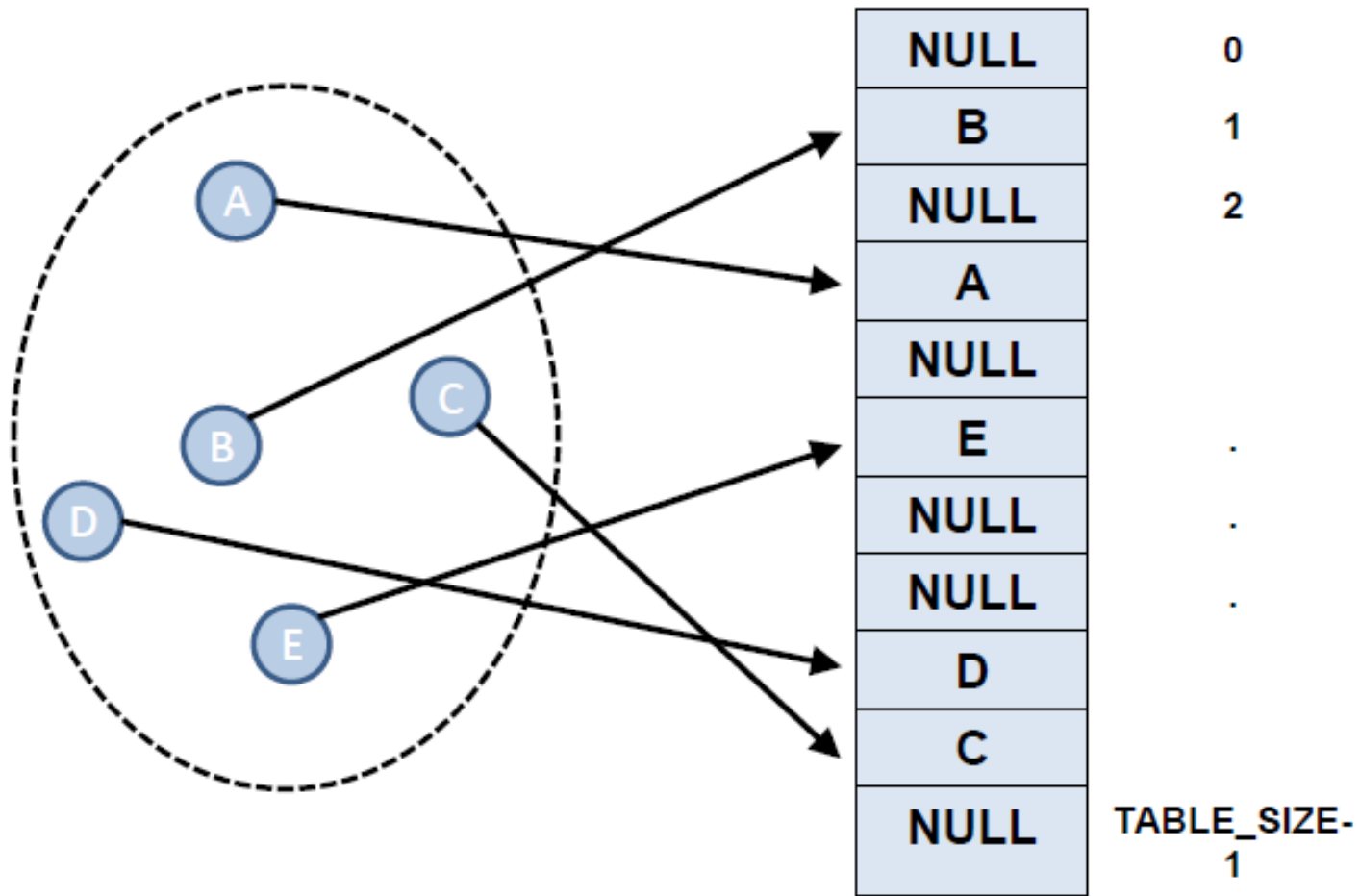


Tabela Hash

- Por que espalhar os elementos melhora a busca?
 - ▣ A tabela permite associar valores a chaves
 - **chave**: parte da informação que compõe o elemento a ser inserido ou buscado na tabela
 - **valor**: é a posição (índice) onde o elemento se encontra no array que define a tabela
 - ▣ Assim, a partir de uma **chave** podemos acessar de forma rápida uma determinada **posição** do array
 - Na média, essa operação tem custo **$O(1)$**

Tabela Hash

□ Vantagens

- Alta eficiência na operação de busca
 - Caso médio é $O(1)$ enquanto o da busca linear é $O(N)$ e a da busca binária é $O(\log_2 N)$
- Tempo de busca é praticamente independente do número de chaves armazenadas na tabela
- Implementação simples

Tabela Hash

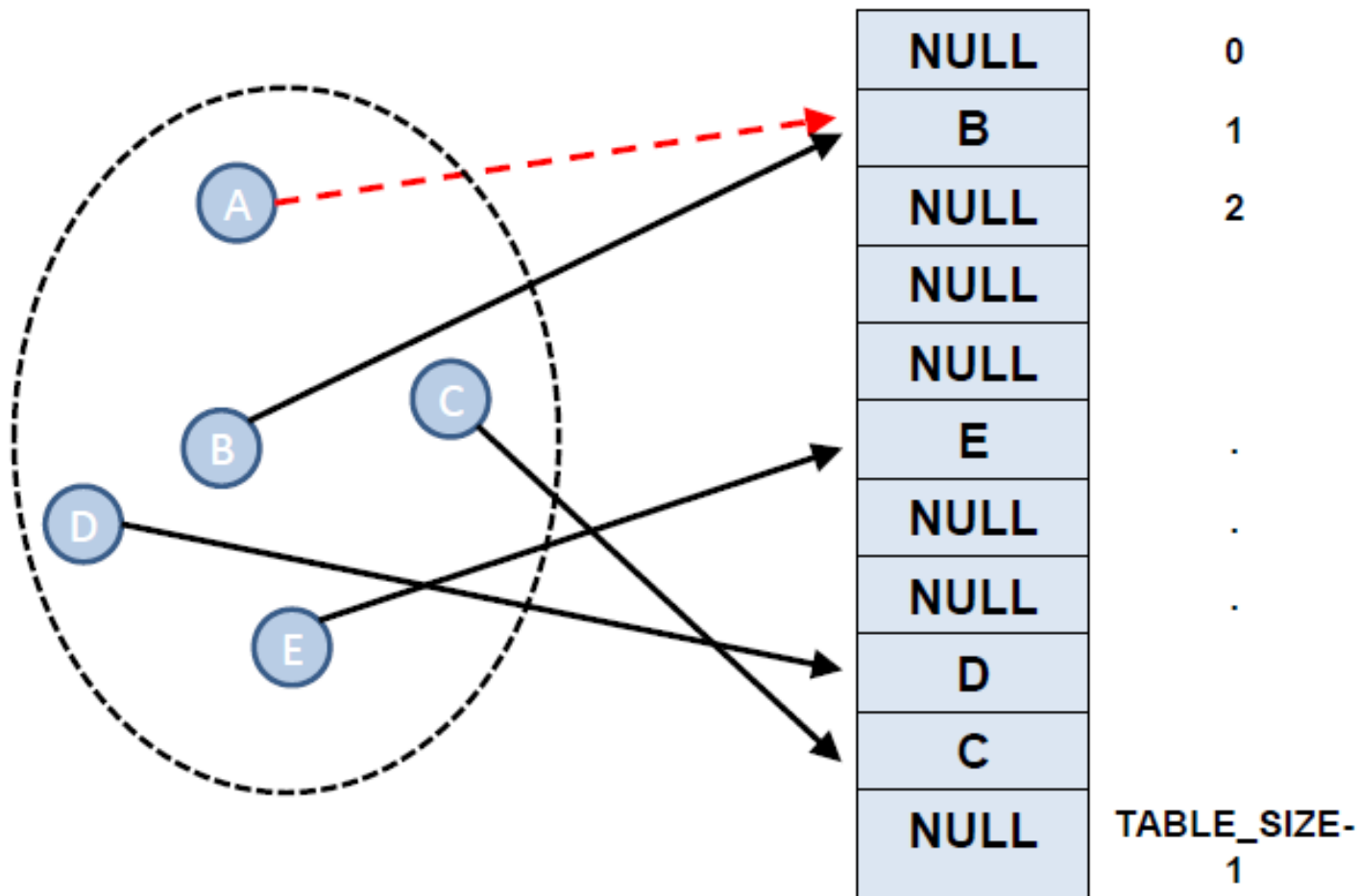
- Infelizmente, esse tipo de implementação também tem suas desvantagens
 - ▣ Alto custo para recuperar os elementos da tabela ordenados pela chave.
 - Nesse caso, é preciso ordenar a tabela
 - ▣ O pior caso é $O(N)$, sendo N o tamanho da tabela
 - Alto número de **colisões**

Tabela Hash

- O que é uma **colisão**?
 - ▣ Uma colisão ocorre quando duas (ou mais) chaves diferentes tentam ocupar a mesma posição na tabela hash.
 - A colisão de chaves não é algo exatamente ruim, é apenas algo indesejável pois diminui o desempenho do sistema.

Tabela Hash

- Exemplo de colisão
 - A** tenta ocupar a posição onde **B** está



Aplicações

- A tabela hash pode ser utilizada para
 - ▣ busca de elementos em base de dados
 - estruturas de dados em memória, bancos de dados e mecanismos de busca na Internet;
 - ▣ verificação de integridade de dados e autenticação de mensagens
 - os dados são enviados juntamente com o resultado da função de hashing
 - Quem receber os dados recalcula a função de hashing usando os dados recebidos e compara o resultado obtido com o que ele recebeu.
 - Resultados diferentes: erro de transmissão

Aplicações

- A tabela hash pode ser utilizada para
 - ▣ armazenamento de senhas com segurança
 - a senha não é armazenada no servidor, mas sim o resultado da função de hashing
 - ▣ implementação da tabela de símbolos dos compiladores
 - ▣ Criptografia
 - MD5 e família SHA (Secure Hash Algorithm).

Implementação da Tabela Hash

- Sua implementação utiliza uma estrutura similar à da [Lista Sequencial Estática](#)
- Utiliza um [array](#) para armazenar os elementos

Desvantagens:

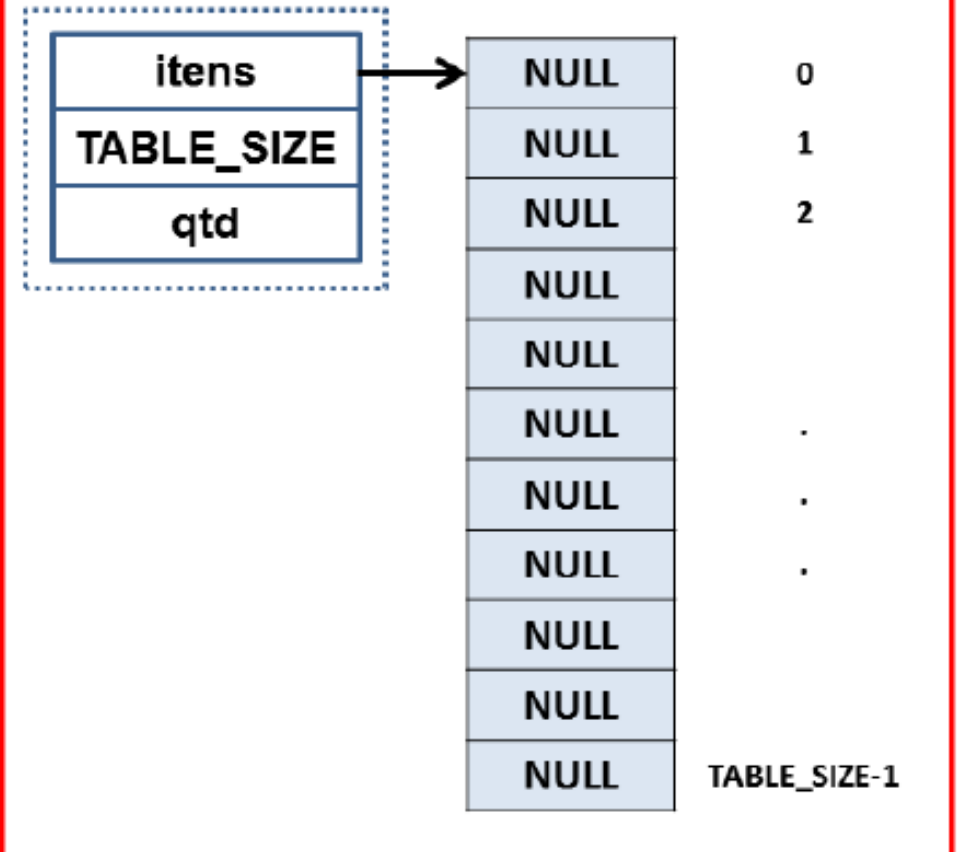
- Necessita que se defina o tamanho do [array](#) previamente
- Isso limita o número de elementos que se pode armazenar

TAD Tabela Hash

```
1 //Arquivo TabelaHash.h
2 struct aluno{
3     int matricula;
4     char nome[30];
5     float n1,n2,n3;
6 };
7 typedef struct hash Hash;
```

```
1 //Arquivo TabelaHash.c
2 #include <stdlib.h>
3 #include <string.h>
4 #include "TabelaHash.h" //inclui o:
5
6 //Definição do tipo Hash
7 struct hash{
8     int qtd, TABLE_SIZE;
9     struct aluno **itens;
10 };
```

Hash *ha



TAD Tabela Hash

□ Importante

- Por questões de desempenho, a tabela irá armazenar apenas o endereço para a estrutura que contém os dados e não os dados em si
- Isso evita o gasto excessivo de memória
- A medida que os elementos são inseridos na tabela, nós realizamos a alocação daquele único elemento

Criando e Destruidendo uma Tabela Hash

Criação da tabela:

Aloca-se espaço para o **array** de ponteiros e os inicializa com **NULL**

Destruição da tabela:

- Envolve percorrer todo o **array** e verificar se existe um elemento para ser desalocado

TAD Tabela Hash

□ Criando a tabela

```
1 //arquivo TabelaHash.h
2 Hash* criaHash(int TABLE_SIZE);
3 //arquivo TabelaHash.c
4 Hash* criaHash(int TABLE_SIZE){
5     Hash* ha = (Hash*) malloc(sizeof(Hash));
6     if(ha != NULL){
7         int i;
8         ha->TABLE_SIZE = TABLE_SIZE;
9         ha->itens = (struct aluno**)
10             malloc(TABLE_SIZE * sizeof(struct aluno));
11         if(ha->itens == NULL){
12             free(ha);
13             return NULL;
14         }
15         ha->qtd = 0;
16         for(i=0; i < ha->TABLE_SIZE; i++)
17             ha->itens[i] = NULL;
18     }
19     return ha;
20 }
21 //programa principal
22 Hash* ha = criaHash(1427);
```

TAD Tabela Hash

□ Destruindo a tabela

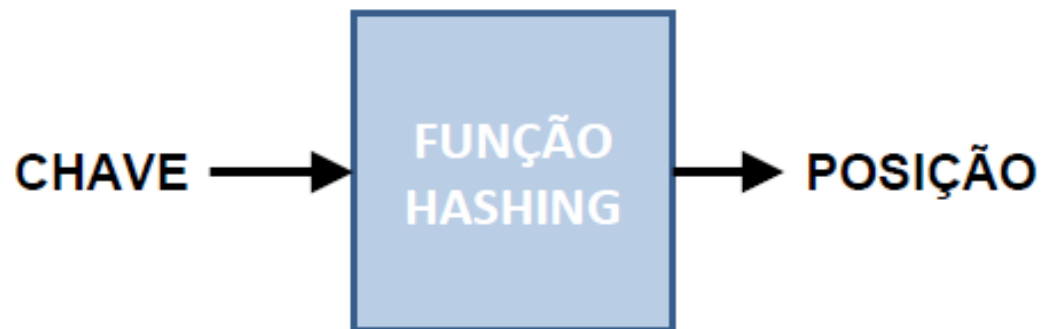
```
1 //arquivo TabelaHash.h
2 void liberaHash(Hash* ha);
3 //arquivo TabelaHash.c
4 void liberaHash(Hash* ha) {
5     if(ha != NULL) {
6         int i;
7         for(i=0; i < ha->TABLE_SIZE; i++) {
8             if(ha->itens[i] != NULL)
9                 free(ha->itens[i]);
10        }
11        free(ha->itens);
12        free(ha);
13    }
14 }
15 //programa principal
16 liberaHash(ha);
```

Tamanho da Tabela Hash

- O ideal é escolher um **número primo** e evitar valores que sejam uma **potência de dois**
 - ▣ Número primo
 - reduz a probabilidade de colisões, mesmo que a função de hashing utilizada não seja muito eficaz
 - ▣ Potência de dois
 - melhora a velocidade, mas pode aumentar os problemas de colisão se estivermos utilizando uma função de hashing mais simples

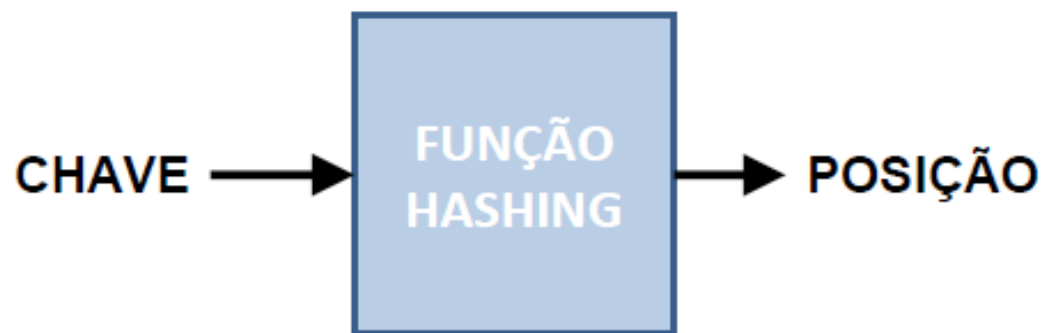
Posição da chave e função de Hashing

- Inserção e busca: é necessário calcular a posição dos dados dentro da tabela.
- Função de Hashing
 - ▣ Calcula a posição a partir de uma chave escolhida a partir dos dados manipulados



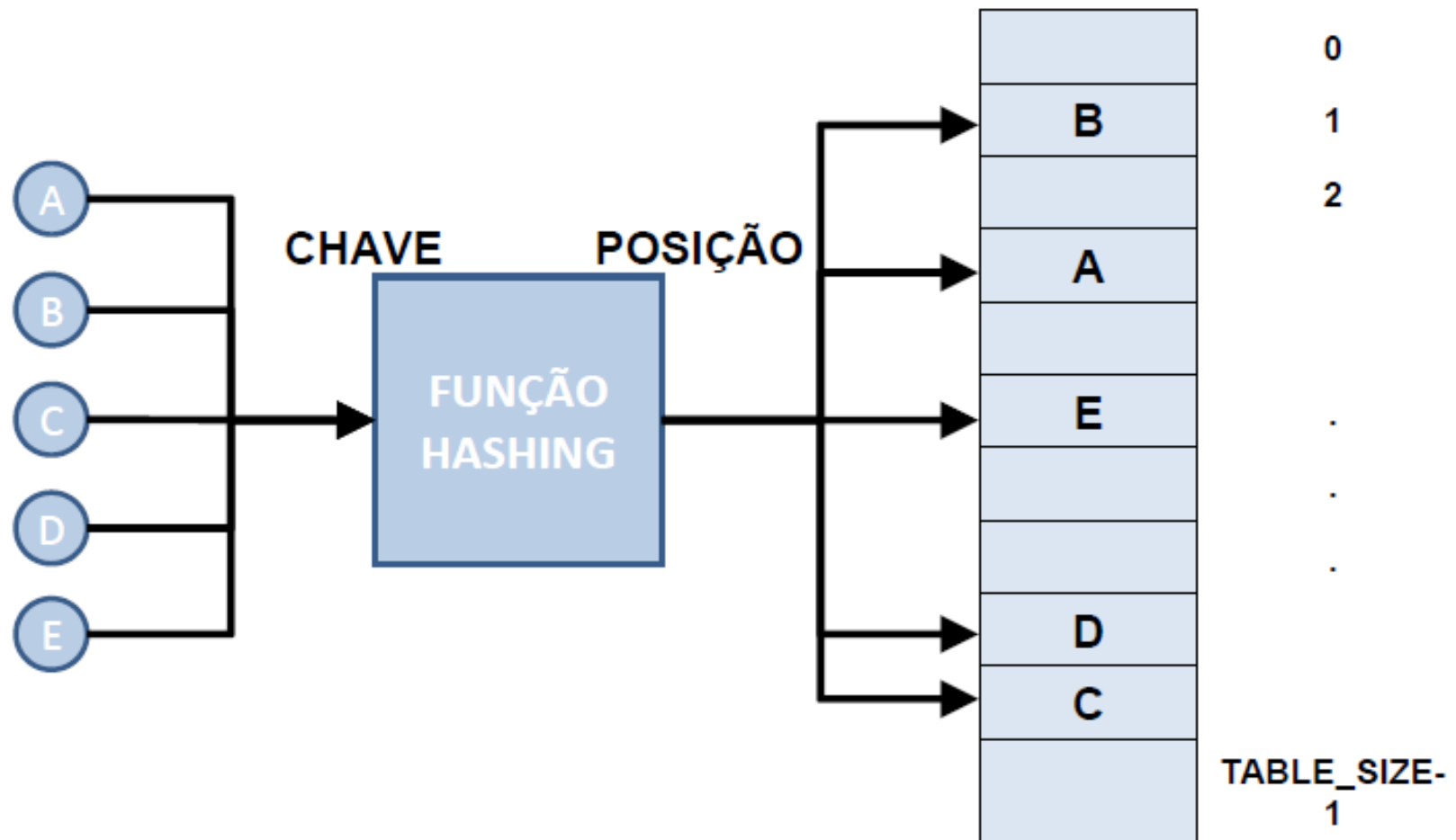
Função de Hashing

- Função de Hashing
 - ▣ É extremamente importante para o bom desempenho da tabela.
 - ▣ Ela é responsável por distribuir as informações de forma equilibrada pela tabela hash



Função de Hashing

Exemplo de funcionamento



Função de Hashing

- Para um bom funcionamento, deve satisfazer as seguintes condições
 - ▣ Ser simples e barata de se calcular
 - ▣ Garantir que valores diferentes produzam posições diferentes
 - ▣ Gerar uma distribuição equilibrada dos dados na tabela
 - Cada posição da tabela tem a mesma chance de receber uma chave (máximo espalhamento)

Função de Hashing

- Sua implementação depende do conhecimento prévio da natureza e domínio da chave a ser utilizada
 - ▣ Exemplo: utilizar apenas três dígitos do número de telefone de uma pessoa para armazená-lo na tabela.
 - Neste caso, seria melhor usar os três últimos dígitos do que os três primeiros, pois os primeiros costumam se repetir com maior frequência e iriam gerar posições iguais na tabela.
 - Assim, o ideal é usar um cálculo diferente de Hash para cada tipo de chave.

Função de Hashing

- Alguns exemplos de função de hashing comumente utilizadas
 - ▣ Método da Divisão
 - ▣ Método da Multiplicação
 - ▣ Método da Dobra

Função de Hashing

- ❑ Método da Divisão
 - ❑ Ou método da congruência linear
 - ❑ Consiste em calcular o **resto da divisão** do valor inteiro que representa o elemento pelo tamanho da tabela, **TABLE_SIZE**
 - Simples e direta
 - A operação de E bit-a-bit (&) com o valor **0x7FFFFFFF** elimina o bit de sinal e evita o risco de ocorrer um **overflow** e obtermos um número negativo

```
9  int chaveDivisao(int chave, int TABLE_SIZE){  
10      return (chave & 0x7FFFFFFF) % TABLE_SIZE;  
11  }
```

Função de Hashing

□ Método da Divisão

▣ Apesar de simples, apresenta alguns problemas.

- Resto da divisão: valores diferentes podem resultar na mesma posição

▣ Exemplo

- O resto da divisão de 11 por 10 e de 21 por 10 são o mesmo valor de posição: 1
- Uma maneira de reduzir esse tipo de problema é utilizar como tamanho da tabela, **TABLE_SIZE**, um número primo

Função de Hashing

□ Método da Multiplicação

▣ Também chamado de **método da congruência linear multiplicativo**

- Usa uma constante fracionária **A**, $0 < A < 1$, para multiplicar o valor da chave que representa o elemento
- Em seguida, a parte fracionária resultante é multiplicada pelo tamanho da tabela para calcular a posição do elemento

```
12 int chaveMultiplicacao(int chave, int TABLE_SIZE){  
13     float A = 0.6180339887; // constante: 0 < A < 1  
14     float val = chave * A;  
15     val = val - (int) val;  
16     return (int) (TABLE_SIZE * val);  
17 }
```

Função de Hashing

- Método da Multiplicação
 - ▣ Exemplo: calcular a posição da chave **123456**, usando a constante fracionária **A = 0,618** e que o tamanho da tabela seja **1024**

```
posição = ParteInteira(TABLE_SIZE * ParteFracionária(chave * A))
```

```
posição = ParteInteira(1024 * ParteFracionária(123456 * 0,618))
```

```
posição = ParteInteira(1024 * ParteFracionária(762950,808))
```

```
posição = ParteInteira(1024 * 0,808)
```

```
posição = ParteInteira(827,392)
```

```
posição = 827
```

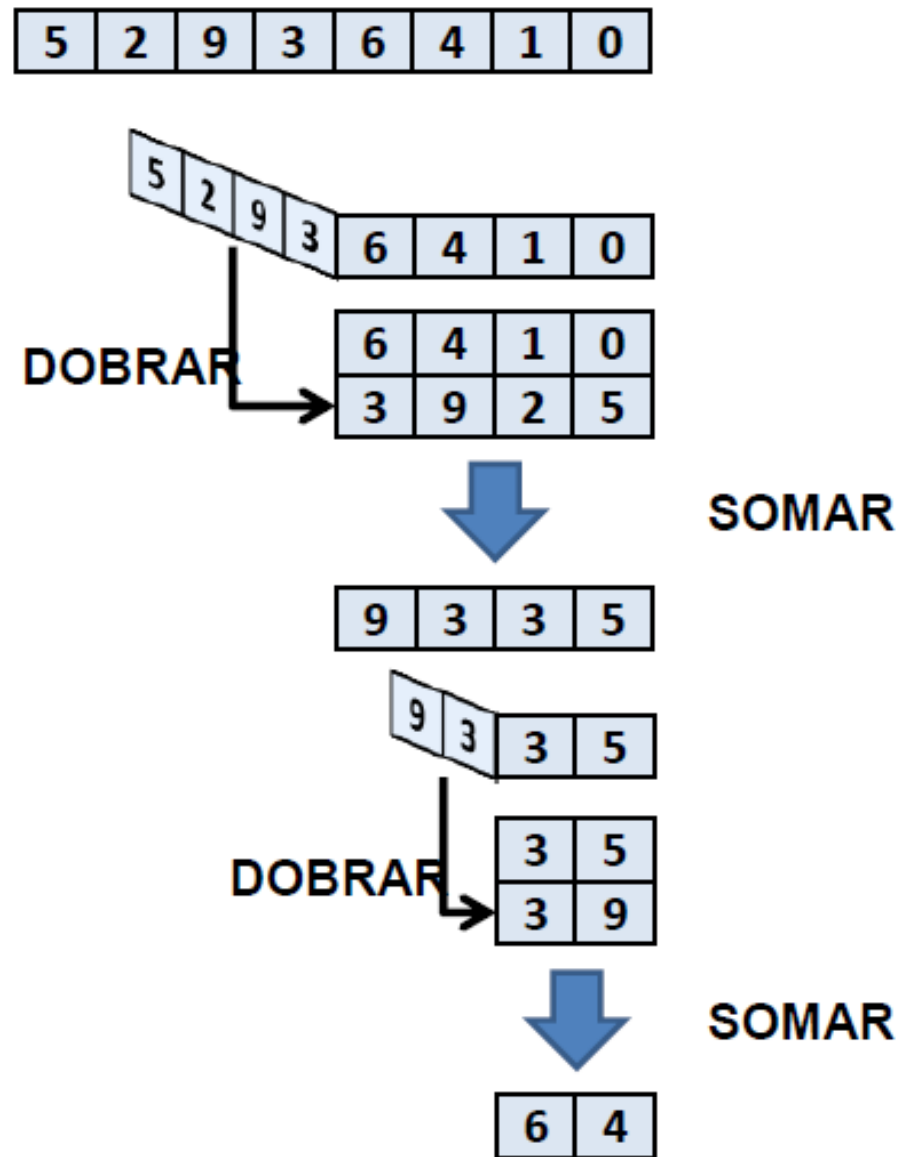

Função de Hashing

- Método da Dobra
 - ▣ Utiliza um esquema de dobrar e somar os dígitos do valor para calcular a sua posição
 - Considera o valor inteiro que representa o elemento como uma sequência de dígitos escritos num pedaço de papel.
 - Enquanto esse valor for maior que o tamanho da tabela, o papel é dobrado e os dígitos sobrepostos são somados, desconsiderando-se as dezenas
 - Note que este processo deve ser repetido enquanto os dígitos formarem um número maior que o tamanho da tabela.

Função de Hashing

□ Método da Dobra

□ Exemplo



Função de Hashing

□ Método da Dobra

▣ Pode ser usado com valores binários

- Utiliza a operação de **OU exclusivo**
- Não se usa as operações de **E** e **OU binário** pois estas produzem resultados menores e maiores, respectivamente, que os operandos

```
9  □int chaveDobra(int chave, int TABLE_SIZE){  
10     int num_bits = 10;  
11     int parte1 = chave >> num_bits;  
12     int parte2 = chave & (TABLE_SIZE-1);  
13     return (parte1 ^ parte2);  
14 }
```

Função de Hashing

- Método da Dobra
 - ▣ No caso de valores binários, a dobra é realizada de k em k bits, o que resulta em um valor de posição entre 0 e 2^{k+1} .
 - Exemplo: queremos calcular a posição do valor 71 (**0001000111** em binário), usando **$k = 5$** :

```
posição = 00010 "OU exclusivo" 00111  
posição = 00101  
posição = 5
```

Função de Hashing

- Tratando uma string como chave
 - ▣ Podemos optar por calcular um valor numérico a partir dessa string
 - Esse valor pode ser facilmente calculado somando os valores ASCII dos caracteres que compõem a string
 - ▣ O resultado pode então ser utilizado como parâmetro para um função de hashing

```
7  □int valorString(char *str){  
8      int i, valor = 7;  
9      int tam = strlen(str);  
10     for(i=0; i < tam; i++)  
11         valor = 31 * valor + (int) str[i];  
12     return valor;  
13 }
```

Função de Hashing

- Tratando uma string como chave
 - ▣ Por que não devemos simplesmente somar os valores ASCII dos caracteres da string?
 - Porque palavras com letras trocadas irão produzir o mesmo valor e, conseqüentemente, uma colisão
 - ▣ Exemplo

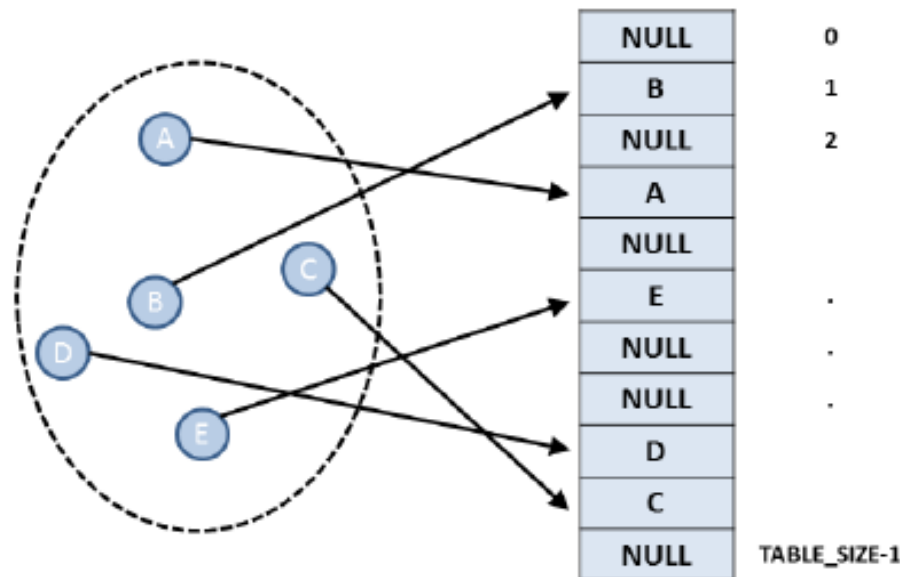
<code>cama:</code>	<code>99</code>	<code>+</code>	<code>97</code>	<code>+</code>	<code>109</code>	<code>+</code>	<code>97</code>	<code>=</code>	<code>402</code>
<code>maca:</code>	<code>109</code>	<code>+</code>	<code>97</code>	<code>+</code>	<code>99</code>	<code>+</code>	<code>97</code>	<code>=</code>	<code>402</code>

Função de Hashing: Inserção e Busca

□ Inserção e busca sem tratamento de colisão

▣ Inserção

- Calcular a **posição** da **chave** no array
- Alocar espaço para os dados
- Armazenar os dados na posição calculada



Função de Hashing: Inserção e Busca

□ Inserção sem tratamento de colisão

```
1 //arquivo TabelaHash.h
2 int insereHash_SemColisao(Hash* ha, struct aluno al);
3 //arquivo TabelaHash.c
4 int insereHash_SemColisao(Hash* ha, struct aluno al){
5     if(ha == NULL || ha->qtd == ha->TABLE_SIZE)
6         return 0;
7     int chave = al.matricula;
8     //int chave = valorString(al.nome);
9     int pos = chaveDivisao(chave, ha->TABLE_SIZE);
10    struct aluno* novo;
11    novo = (struct aluno*) malloc(sizeof(struct aluno))
12    if(novo == NULL)
13        return 0;
14    *novo = al;
15    ha->itens[pos] = novo;
16    ha->qtd++;
17    return 1;
18 }
19 //programa principal
20 int x = insereHash_SemColisao(ha, al);
```

Função de Hashing: Inserção e Busca

- Inserção e busca sem tratamento de colisão
 - ▣ Busca
 - Calcular a **posição** da **chave** no array
 - Verificar se há dados na **posição** calculada
 - Retornar os dados

Função de Hashing: Inserção e Busca sem Tratamento de Colisão

□ Busca sem tratamento de colisão

```
1 //arquivo TabelaHash.h
2 int buscaHash_SemColisao(Hash* ha, int mat,
3                           struct aluno* al);
4 //arquivo TabelaHash.c
5 int buscaHash_SemColisao(Hash* ha, int mat,
6                           struct aluno* al){
7     if(ha == NULL)
8         return 0;
9
10    int pos = chaveDivisao(mat, ha->TABLE_SIZE);
11    if(ha->itens[pos] == NULL)
12        return 0;
13    *al = *(ha->itens[pos]);
14    return 1;
15 }
16 //programa principal
17 int x = buscaHash_SemColisao(ha, mat, &al);
```

Hashing Universal

- Função de hashing está sujeita ao problema de gerar posições iguais para chaves diferentes
 - ▣ Por se tratar de uma função determinística, ela pode ser manipulada de forma indesejada.
 - ▣ Conhecendo a função de hashing, pode-se escolher as chaves de entrada de modo que todas colidam, diminuindo o desempenho da tabela na busca para $O(N)$

Hashing Universal

- Hashing universal é uma estratégia que busca minimizar esse problema de colisões
 - ▣ Basicamente, devemos escolher aleatoriamente (em tempo de execução) a **função de hashing** que será utilizada.
 - ▣ Para tanto, construímos um conjunto (ou família) de **funções de hashing**

Hashing Universal

- Existem várias maneiras diferentes de construir uma família de funções de hashing.
 - ▣ Uma família de funções pode ser facilmente obtida da seguinte forma:
 - Escolha um número primo p . Ele deve ser maior do que qualquer chave k a ser inserida.
 - p também deve ser maior do que o tamanho da tabela, **TABLE_SIZE**
 - Escolha, aleatoriamente, dois números inteiros, a e b , de tal modo que $0 < a \leq p$ e $0 \leq b \leq p$

Hashing Universal

- Dados os valores **p**, **a**, e **b**, definimos a função de hashing universal como sendo
 - ▣ $h(k)_{a,b} = ((ak + b) \% p) \% \text{TABLE_SIZE}$
 - Esse tipo de função de hashing universal permite o tamanho da tabela, **TABLE_SIZE**, não seja necessariamente primo
 - Além disso, como existem **p-1** valores diferentes para o valor de **a** e **p** valores possíveis para **b**, é possível gerar **p(p-1)** funções de hashing diferentes.

Hashing imperfeito e perfeito

- A depender do tamanho da tabela, **TABLE_SIZE**, e dos valores inseridos, uma função de hashing pode ser definida como
 - ▣ Hashing imperfeito
 - ▣ Hashing perfeito

Hashing imperfeito e perfeito

- Hashing imperfeito
 - ▣ Para duas chaves diferentes a saída da função de hashing é a mesma posição na tabela
 - ▣ Ou seja, podem ocorrer colisões das chaves
 - A colisão de chaves não é algo exatamente ruim, é apenas algo indesejável pois diminui o desempenho do sistema
 - De modo geral, muitas tabelas hash fazem uso de alguma outra estrutura de dados para lidar com o problema da colisão, como veremos adiante.

Hashing imperfeito e perfeito

□ Hashing perfeito

▣ Nunca ocorre colisão

- Chaves diferentes irão sempre produzir posições diferentes

▣ No pior caso, as operações de busca e inserção são sempre executadas em tempo constante, **$O(1)$** .

- É utilizado onde a colisão não é tolerável
- Trata-se de um tipo de aplicação muito específica, por exemplo, o **conjunto de palavras reservadas de uma linguagem de programação**. Nesse caso, conhecemos previamente o conteúdo a ser armazenado na tabela

Tratamento de Colisões

- Mundo ideal
 - ▣ Hashing perfeito
 - Função de hashing irá sempre fornecer posições diferentes para cada uma das chaves inseridas
- Mundo real
 - ▣ Independente da função de hashing utilizada, a mesma vai retornar a mesma **posição** para duas **chaves** diferentes: **colisão!**

Tratamento de Colisões

- A criação de uma tabela hash consiste de duas coisas
 - ▣ uma **função de hashing**
 - ▣ uma **abordagem para o tratamento de colisões**

Tratamento de Colisões

- Uma escolha adequada do **tamanho da tabela** pode minimizar as colisões
 - ▣ Colisões ocorrerem porque temos mais chaves para armazenar do que o tamanho da tabela suporta
 - ▣ Não há espaço suficiente para todas as chaves

Tratamento de Colisões

- Uma escolha adequada da **função de hashing** pode minimizar as colisões
 - ▣ Escolher uma função que produza um espalhamento uniforme das chaves reduz o número de colisões
 - Infelizmente, não se pode garantir que as funções de hashing possuam um bom potencial de espalhamento por que as colisões também são uniformemente distribuídas.
 - Colisões são teoricamente inevitáveis

Tratamento de Colisões

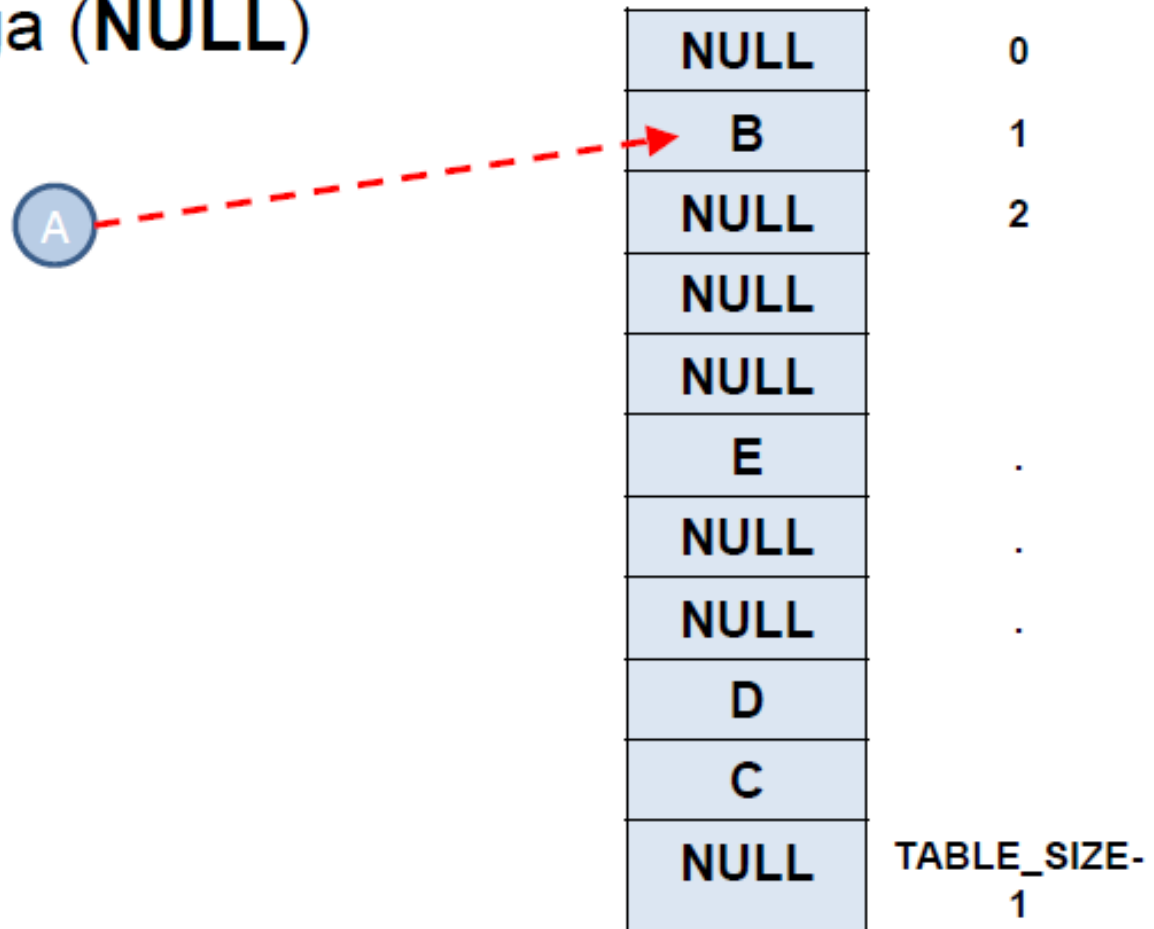
- Colisões são teoricamente inevitáveis. Por isso, devemos sempre ter uma abordagem para tratá-las.
 - ▣ Existem diversas formas de se tratar a colisão
 - ▣ Duas técnicas muito comuns
 - endereçamento aberto
 - encadeamento separado

Endereçamento A berto

- Definição
 - ▣ Também conhecido como *open addressing* ou *rehash*
 - ▣ No caso de um **colisão**, percorrer a tabela hash buscando por uma **posição** ainda não ocupada
 - ▣ Os elementos são armazenados na própria tabela hash
 - Evita o uso de listas encadeadas

Endereçamento A berto

- **A** tenta ocupar a posição de **B**
 - ▣ Devemos percorrer a tabela até achar uma posição vaga (**NULL**)



Endereçamento A berto

□ Vantagens

- Maior número de posições na tabela para a mesma quantidade de memória usada no **encadeamento separado**
- A memória utilizada para armazenar os ponteiros da lista encadeada no **encadeamento separado** pode ser aqui usada para aumentar o tamanho da tabela, diminuindo o número de colisões

Endereçamento A berto

□ Vantagens

- Busca é realizada dentro da própria tabela
 - Recuperação mais rápida de elementos
- Voltada para aplicações com restrições de memória
- Ao invés de acessarmos ponteiros extras, calculamos a sequência de posições a serem armazenadas.

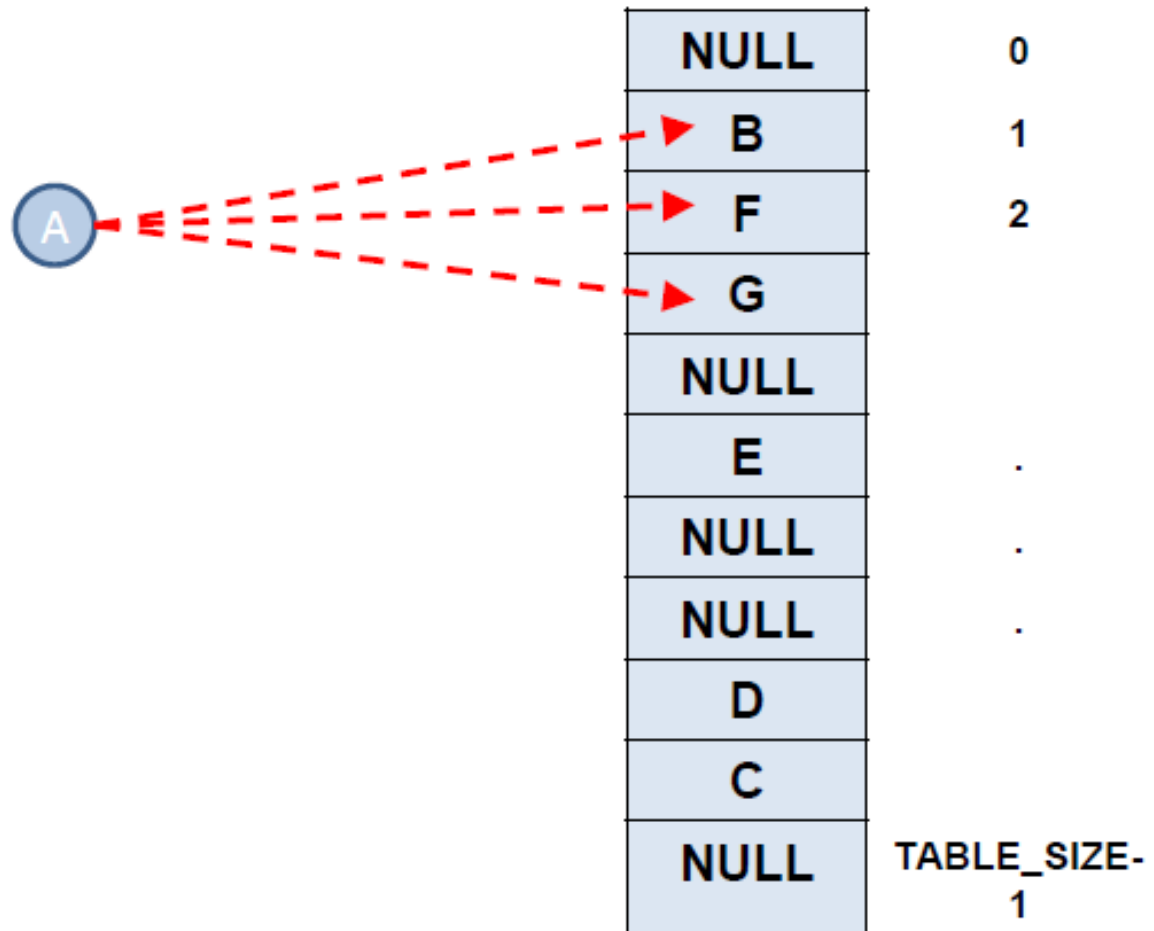
Endereçamento A berto

- Desvantagens

- ▣ Maior esforço de processamento no cálculo das posições
- ▣ Esse esforço maior se deve ao fato de que, quando uma colisão ocorre, devemos calcular uma nova posição da tabela
 - Colisões sucessivas
 - Quando isso ocorre, o custo da inserção se torna $O(N)$

Endereçamento A berto

- Se apenas percorrermos o array, teremos colisões sucessivas



Endereçamento A berto

- Para a realização do cálculo da nova posição após a colisão, existem três estratégias muito utilizadas
 - ▣ Sondagem linear
 - ▣ Sondagem quadrática
 - ▣ Duplo hash

Endereçamento A berto

- Sondagem linear
 - ▣ Também conhecida como **tentativa linear**, **espalhamento linear** ou **rehash linear**
 - ▣ Tenta espalhar os elementos de forma sequencial a partir da posição calculada utilizando a função de hashing

Endereçamento A berto

□ Sondagem linear

▣ Funcionamento


- Primeiro elemento ($i = 0$) é colocado na **posição** obtida pela **função de hashing: pos**
- Segundo elemento (colisão) é colocado na posição **pos+1**
- Terceiro elemento (nova colisão) é colocado na posição **pos+2**

```
16 int sondagemLinear(int pos, int i, int TABLE_SIZE) {  
17     return ((pos + i) & 0x7FFFFFFF) % TABLE_SIZE;  
18 }
```

Endereçamento A berto

□ Sondagem linear

NULL	0	CHAVE	POSIÇÃO	INSERÇÃO	E	0
NULL	1	A	2	Posição 2 vazia. Insere elemento	NULL	1
NULL	2	B	6	Posição 6 vazia. Insere elemento	A	2
NULL	3	C	2	Posição 2 ocupada, procura na próxima posição: 3	C	3
NULL	4			Posição 3 vazia. Insere elemento	NULL	4
NULL	5	D	10	Posição 10 vazia. Insere elemento	NULL	5
NULL	6	E	10	Posição 10 ocupada, procura na próxima posição. Como a posição 10 é a última, volta para o início: 0	B	6
NULL	7				NULL	7
NULL	8				NULL	8
NULL	9				NULL	9
NULL	10				D	10



Endereçamento A berto

- Sondagem linear
 - ▣ Estratégia simples
 - ▣ Apresenta um problema conhecido como **agrupamento primário**
 - A medida que a tabela hash fica cheia, o tempo para incluir ou buscar um elemento aumenta
 - A medida que os elementos são inseridos surgem longas sequências de posições ocupadas
 - A ocorrência desses agrupamentos aumenta o tempo de pesquisa, diminuindo o desempenho

Endereçamento A berto

- Sondagem quadrática
 - ▣ Também conhecida como tentativa quadrática, espalhamento quadrático ou rehash quadrático
 - ▣ Tenta espalhar os elementos utilizando uma equação do segundo grau
 - ▣ Exemplo
 - $pos + (c_1 * i) + (c_2 * i^2)$
 - **pos** é a posição obtida pela função de hashing
 - **i** é tentativa atual
 - **c₁** e **c₂** são os coeficientes da equação

Endereçamento A berto

□ Sondagem quadrática

▣ Funcionamento

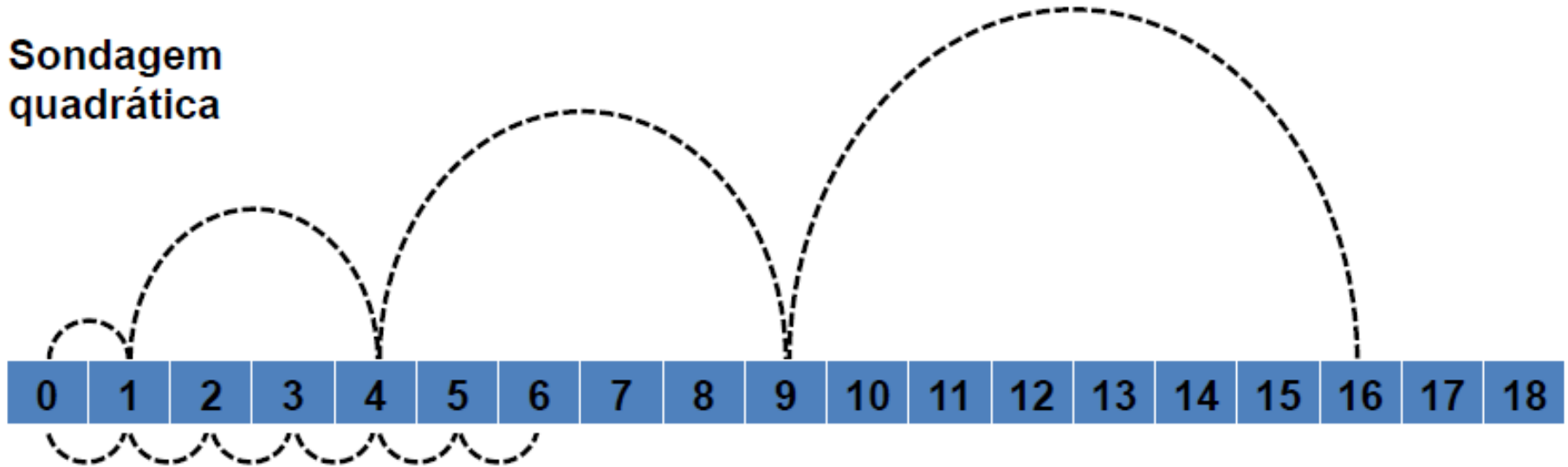
- Primeiro elemento ($i = 0$) é colocado na **posição** obtida pela **função de hashing: pos**
- Segundo elemento (colisão) é colocado na posição **$\text{pos} + (c_1 * 1) + (c_2 * 1^2)$**
- Terceiro elemento (nova colisão) é colocado na posição **$\text{pos} + (c_1 * 2) + (c_2 * 2^2)$**

```
11 int sondagemQuadratica(int pos, int i, int TABLE_SIZE) {  
12     pos = pos + 2*i + 5*i*i;  
13     return (pos & 0x7FFFFFFF) % TABLE_SIZE;  
14 }
```

Endereçamento A berto

□ Sondagem quadrática

Sondagem quadrática



Sondagem linear

Endereçamento A berto

- Sondagem quadrática
 - ▣ Resolve o problema de agrupamento primário
 - ▣ Porém, gera outro problema conhecido como **agrupamento secundário**
 - Todas as chaves que produzam a mesma posição inicial também produzem as mesmas posições na sondagem quadrática
 - Felizmente, a degradação produzida pelos agrupamentos secundários ainda é menor que a produzida pelos agrupamentos primários

Endereçamento A berto

- Duplo hash
 - ▣ Também conhecida como espalhamento duplo
 - ▣ Tenta espalhar os elementos utilizando duas funções de hashing:
 - a **primeira** função de hashing, **H1**, é utilizada para calcular a posição inicial do elemento
 - a **segunda** função de hashing, **H2**, é utilizada para calcular os deslocamentos em relação a posição inicial (no caso de uma colisão)

Endereçamento A berto

- Duplo hash
 - A posição de um novo elemento na tabela hash é obtida como sendo
 - $H1 + i * H2$
 - onde i é tentativa atual de inserção do elemento
 - É necessário que as duas funções de hashing sejam diferentes.
 - A segunda função de hashing não pode resultar em um valor igual a ZERO pois, neste caso, não haveria deslocamento

Endereçamento A berto

□ Duplo hash

▣ Funcionamento

- Primeiro elemento ($i = 0$) é colocado na **posição** obtida por **H1**
- Segundo elemento (colisão) é colocado na posição **$H1 + 1 * H2$**
- Terceiro elemento (nova colisão) é colocado na posição **$H1 + 2 * H2$**

```
11  □ int duploHash(int H1, int chave, int i, int TABLE_SIZE)
12  |     int H2 = chaveDivisao(chave, TABLE_SIZE-1) + 1;
13  |     return ((H1 + i*H2) & 0x7FFFFFFF) % TABLE_SIZE;
14  | }
```

Tabela Hash: Inserção e Busca com Tratamento de Colisão

□ Inserção e busca com tratamento de colisão

▣ Inserção

- Calcular a **posição** da **chave** no array
- Recalcular a **posição** enquanto houver **colisão** (limitar o número de tentativas)
- Alocar espaço para os dados
- Armazenar os dados na posição calculada

Tabela Hash: Inserção e Busca com Tratamento de Colisão

□ Inserção com tratamento de colisão

```
2  int insereHash_EnderAberto(Hash* ha, struct aluno al){
3      if(ha == NULL || ha->qtd == ha->TABLE_SIZE)
4          return 0;
5      int chave = al.matricula;
6      int i, pos, newPos;
7      pos = chaveDivisao(chave, ha->TABLE_SIZE);
8      for(i=0; i < ha->TABLE_SIZE; i++){
9          newPos = sondagemLinear(pos, i, ha->TABLE_SIZE);
10         if(ha->itens[newPos] == NULL){
11             struct aluno* novo;
12             novo = (struct aluno*)
13                 malloc(sizeof(struct aluno));
14             if(novo == NULL)
15                 return 0;
16             *novo = al;
17             ha->itens[newPos] = novo;
18             ha->qtd++;
19             return 1;
20         }
21     }
22     return 0;
23 }
24 //programa principal
25 int x = insereHash_EnderAberto(ha, al);
```

Tabela Hash: Inserção e Busca com Tratamento de Colisão

□ Inserção e busca com tratamento de colisão

▣ Busca

- Calcular a **posição** da **chave** no array
- Verificar se há dados na **posição** calculada e se esses dados combinam com a **chave**
- Recalcular a **posição** enquanto os dados forem diferentes da **chave**
- Retornar os dados

Tabela Hash: Inserção e Busca com Tratamento de Colisão

□ Busca com tratamento de colisão

```
1 //arquivo TabelaHash.h
2 int buscaHash_EnderAberto (Hash* ha, int mat,
3                             struct aluno* al);
4 //arquivo TabelaHash.c
5 int buscaHash_EnderAberto (Hash* ha, int mat,
6                             struct aluno* al) {
7     if (ha == NULL)
8         return 0;
9     int i, pos, newPos;
10    pos = chaveDivisao (mat, ha->TABLE_SIZE);
11    for (i=0; i < ha->TABLE_SIZE; i++) {
12        newPos = sondagemLinear (pos, i, ha->TABLE_SIZE);
13        if (ha->itens[newPos] == NULL)
14            return 0;
15        if (ha->itens[newPos]->matricula == mat) {
16            *al = *(ha->itens[newPos]);
17            return 1;
18        }
19    }
20    return 0;
21 }
22 //programa principal
23 int x = buscaHash_EnderAberto (ha, mat, &al);
24
```

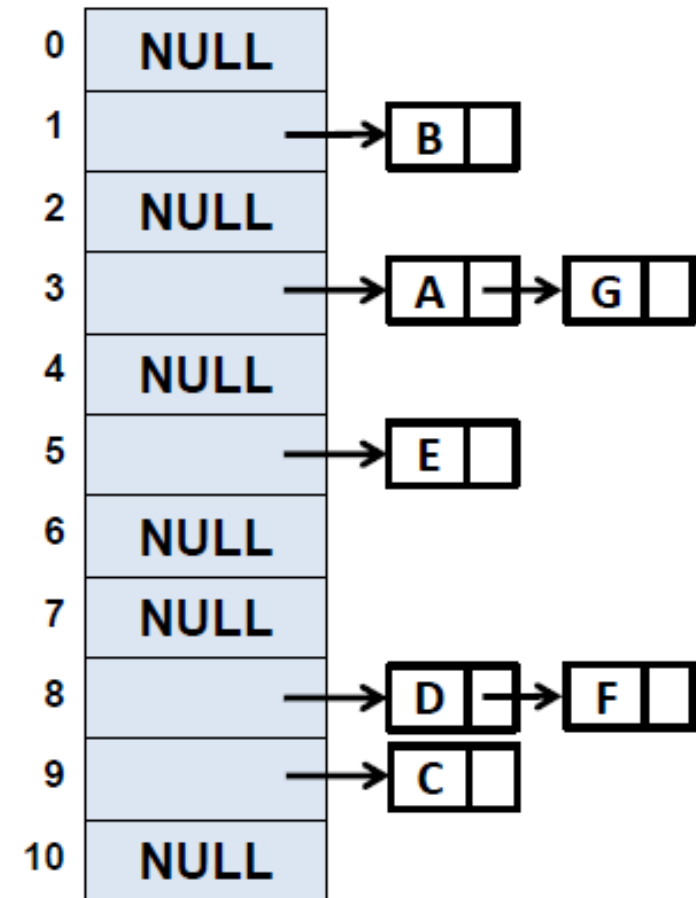
Encadeamento Separado

- Também conhecido como *separate chaining*
 - ▣ Não procura por posições vagas (valor **NULL**) dentro do array que define a tabela
 - ▣ Armazena dentro de cada posição do array o início de uma lista dinâmica encadeada
 - É dentro dessa lista que serão armazenadas as colisões (elementos com chaves iguais) para aquela posição do array

Encadeamento Separado

Exemplo

CHAVE	POSIÇÃO
A	3
B	1
C	9
D	8
E	5
F	8
G	3



Encadeamento Separado

□ Características

- ▣ A lista dinâmica encadeada mantida em cada posição da tabela pode ser ordenada ou não
- ▣ Lista não ordenada
 - Inserção tem complexidade $O(1)$ no pior caso: basta inserir o elemento no início da lista.
 - Busca tem complexidade $O(M)$ no pior caso: busca linear
- ▣ Desvantagem
 - Quantidade de memória consumida: gastamos mais memória para manter os ponteiros que ligam os diferentes elementos dentro de cada lista

Referências

Backes, André. Slides de aulas: Tabela Hash.