N_EST DAD_A8 – Texto de Apoio

Site: EAD Mackenzie
Tema: ESTRUTURA DE DADOS {TURMA 03A} 2023/1
Livro: N_EST DAD_A8 – Texto de Apoio

Impresso por: FELIPE BALDIM GUERRA .
Data: quarta, 10 mai 2023, 00:20

Índice

TABELAS HASH

Conceituando

As Tabelas *Hash* (*Hash Tables* – tabelas de espalhamento) fornecem métodos para armazenar e recuperar registros a partir de um arranjo de elementos. Com ela, é possível que você insira, exclua e busque informações com base em um valor de chave. Quando adequadamente aplicadas, essas operações podem ser realizadas em tempo constante.

Uma Tabela Hash nada mais é que um vetor capaz de armazenar conjuntos de dados do tipo Chave, Valor Associado (K,V).

Situação exemplo 1:

Imagine que uma loja de e-commerce tenha atualmente 50.000 produtos à venda dentro de um catálogo de quase 100.000 itens. Cada produto tem um código de cinco dígitos decimais associado a ele. Veja alguns exemplos:

Código (chave)	nome produto (valor associado)
01456	Batedeira
01578	Processador de alimentos
.....	
34508	Liquidificador
41238	Conjunto de refratários
98980	Jogo de facas

Como armazenar esses dados em um vetor? Uma solução seria criar um vetor de 100.000 posições e armazenar cada item utilizando o próprio código (chave) como índice do vetor.

0	....	1456	....	1578	...	34508	...	98980	...	100000
		Batedeira		Processador		Liquidificador		Jogo de facas		

- Esse vetor é conhecido como Tabela Hash (TH).
- Esse vetor ficará com muitas posições “vazias” (desperdício de espaço).
- A busca e a inserção são extremamente rápidas (tempo constante – O(1)), uma vez que conseguimos acessar qualquer elemento com o mesmo tempo, independentemente da quantidade de elementos ou posições ocupadas.

Veja que, neste exemplo, temos uma tabela muito grande com poucos dados armazenados. O ideal é que todas as posições de uma TH sejam ocupadas. E quem determinará quais posições serão ocupadas é a função Hash.

Conceito de Função Hash

O hashing realiza um cálculo de uma chave de pesquisa K de maneira que se destina a identificar a posição da TH que contém o registro com a chave K.

Figura 1 – Esquema de armazenamento de dados na Tabela Hash

Fonte: Elaborada pela autora.

A função que faz este cálculo é chamada de **função hash** e ela será representada pela letra h. O número de entradas em uma TH será denotado pela variável M, e elas serão numeradas de 0 a M-1. A função de espalhamento ou função hashing (hash function): transforma cada chave em um índice da tabela hash. A função hashing sempre responde à pergunta: "Em qual posição da tabela hash devo colocar esta chave?".

A função hashing espalha as chaves pela tabela hash, uma vez que associa um valor hash (hash value), entre 0 e M−1, para cada chave.

O objetivo de um sistema de hashing é organizar as coisas de tal forma que, para qualquer valor da chave K e algumas funções hash h, i = h(K) é uma entrada na tabela de tal forma que 0 <= i < M. Assim, nós temos a chave do registro armazenado no TH[i] igual a K.

0	**1**	**2**	**3**	**.....**	**M-1**
reg	**reg**	**reg**	**reg**	**.....**	**reg**
h(K)=0	**h(K)=1**	**h(K)=2**	**h(K)=3**		**h(K)=M-1**

Como vimos, uma tabela de dispersão ou tabela hash (*hash table*) é um vetor onde cada uma de suas posições armazena zero, uma, ou mais chaves (e valores associados).

O que muitas vezes acontece é que uma posição do vetor acaba armazenando mais de um valor. Nesse caso, chamamos a entrada da TH de bucket. Parâmetros importantes:

- M ➔ número de posições na tabela hash.
- N ➔ número de chaves que serão armazenadas.
- α = N/M ➔ fator de carga (*load factor*) – o fator de carga determina a média de ocupação de cada bucket e determina se a TH precisa aumentar sua capacidade, ou não.

Exemplo:

- Na situação exemplo 1, M = 100.000, N = 50.000, α <1 e a função hash é o próprio código do produto ➔ h(k) = código produto. Nesse caso, significa que cada entrada da TH armazena menos que uma (K,V).

Colisões

E o que fazer quando existem mais valores a serem armazenados do que espaços disponíveis na TH? E se eu precisar armazenar 1000 valores no intervalo de 0 a 65.535?

Opção 1 ➔ Criar um vetor de 65.535 elementos e ocupar apenas 1000, armazenando os valores de acordo com a função *hash* h(k)=k. Nesse caso, muitas entradas ficarão vazias! Fator da carga = 1000/65535 = 0,01.

Opção 2 ➔ Usar uma função *hash* h de forma que seja utilizada uma TH menor, em que uma única entrada da tabela consegue armazenar mais de um valor de chave.

Situação exemplo 2:

Imagine que todos os produtos do exemplo anterior sejam, agora, armazenados em uma Lista em ordem alfabética:

Nome produto (chave)	Código (valor associado)
Batedeira britânia	01456
Conjunto de refratários	41238
.....	
Jogo de facas	98980
Liquidificador	34508
Processador de alimentos	01578

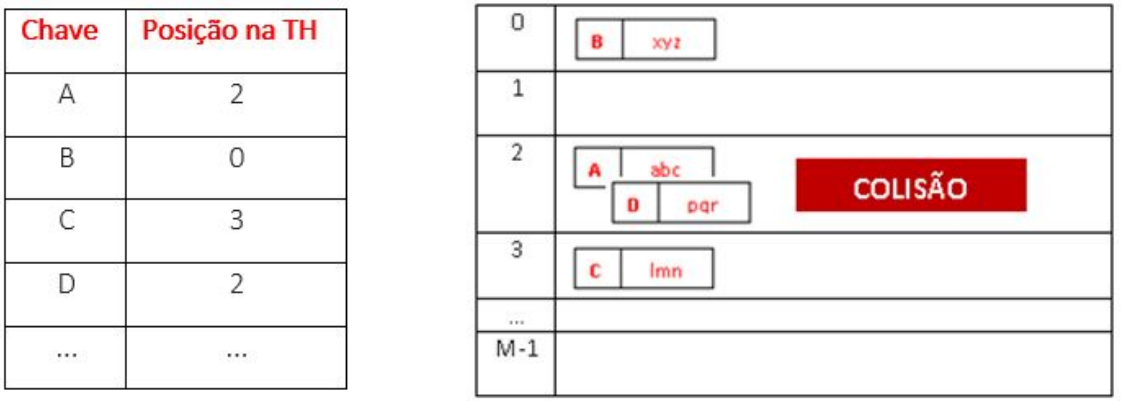
Para acelerar as buscas, a lista foi dividida em 26 partes: os produtos que começam com a letra “A”, os produtos que começam com a letra “B” etc.

- Nesse caso, o vetor de 26 posições é a Tabela Hash.
- Cada posição do vetor “aponta” para o início dessas listas.

- M = 26, N = 50.000, α > 1 e a função hash é o primeiro caractere do nome do produto ➔ h(k) = nome[0].

A função de hashing produz uma *colisão* quando duas chaves diferentes têm o mesmo valor hash e, portanto, são levadas para a mesma posição da tabela hash. Veja a Figura 2 que apresenta quatro chaves e suas respectivas posições na TH (calculados aleatoriamente) e identifica uma colisão na posição 2 da TH:

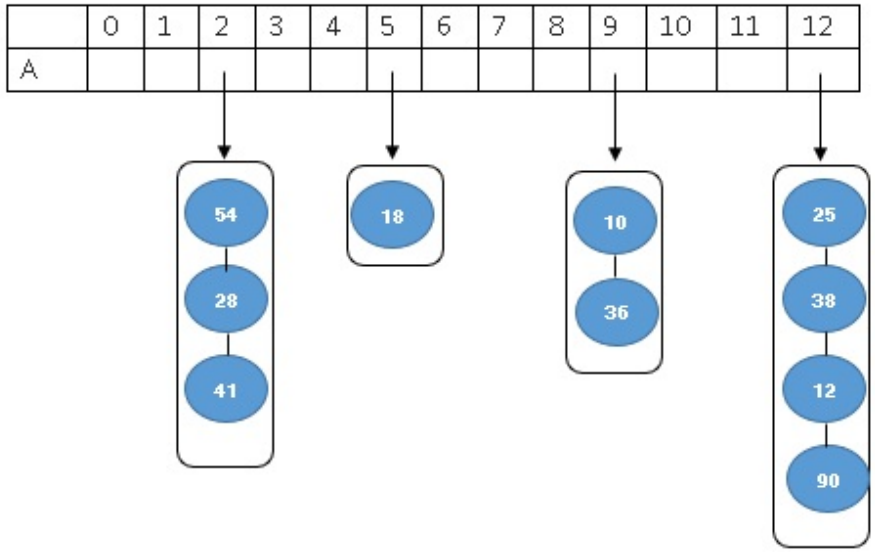
Figura 2 – Exemplo de colisão



Adaptado de: Sedgewick (2020).

As colisões ocorrem quando dois registros são direcionados para a mesma entrada na tabela *hash* (TH). Para resolver o problema da colisão, muitas vezes, a Tabela Hash é combinada com outra estrutura de dados. Isso ocorre da seguinte forma: cada posição da Tabela pode armazenar uma Lista, uma Árvore ou outra estrutura de dados em vez de apenas um elemento. Assim, conseguimos armazenar mais de um elemento na mesma posição da TH. Veja a Figura 3 abaixo, que apresenta uma TH com M = 13 e em alguns buckets que dão acesso à uma lista.

Figura 3 – Exemplo de colisão



Fonte: Elaborada pela autora.

Algumas observações importantes:

- Uma TH deve armazenar registros de forma a não desperdiçar espaço.
- Para equilibrar o tempo e a eficiência do espaço, a TH deve ter capacidade para armazenar cerca de metade do total de elementos.
- A diferença entre usar uma boa função hash e uma função hash ruim é percebida, na prática, no número de registros que devem ser examinados na busca ou na inserção na tabela.
- O ideal é escolher uma função hash que mapeia chaves de maneira que faz com que cada entrada na tabela hash tenha igual probabilidade de ser preenchida para as chaves que estão sendo usadas.
- Deve-se selecionar uma função hash que distribua uniformemente o intervalo de chaves por meio da tabela hash, evitando oportunidades óbvias para o agrupamento.

E qual é a relação do Fator de Carga com as colisões? Quanto menor for o Fator de Carga, menor é a chance de colisão acontecer. Veja abaixo a Tabela 1 que apresenta uma comparação entre o fator de carga e a probabilidade de colisão:

Tabela 1 – Fator de carga vs. chance de colisão para TH com M = 500.000

Chave	Chance de colisão	Fator de Carga (N/M)
1000	0,995%	0,002
2000	3,918%	0,004
4000	14,772%	0,008
6000	30,206%	0,012
8000	47,234%	0,016
10000	63,171%	0,020
12000	76,269%	0,024
14000	85,883%	0,028
16000	92,248%	0,032
18000	96,070%	0,036
20000	98,160%	0,040
22000	99,205%	0,044

Fonte: Elaborada pela autora.

Exemplos de Função Hash

Modular

Usada, normalmente, quando as chaves são números inteiros positivos e armazenamos o resto da divisão da chave por M. Veja uma função *hash* modular para M = 16:

```
int h(int x) {
  return x % 16;
}
```

Repare que a função acima recebe como parâmetro um valor inteiro (a chave que pretendemos armazenar na TH). Esse valor é dividido por 16 e o resto da divisão é retornado pela função.

Vamos armazenar algumas chaves na TH abaixo (M = 16) de acordo com essa função. Os valores são: 28, 130 e 57. Faremos as divisões por 16 e armazenaremos os restos.

28 % 16 = 12 (resto)

130 % 16 = 2 (resto)

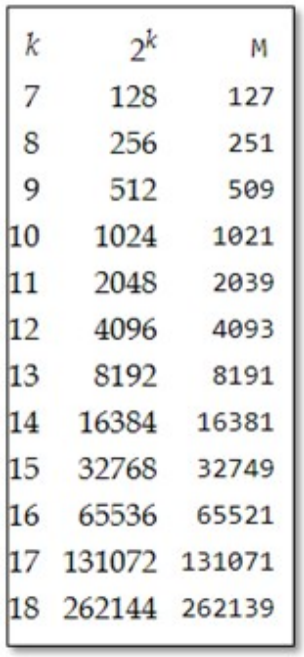
57 % 16 = 9 (resto)

Agora, armazenaremos as chaves na TH de acordo com os restos obtidos. Veja:

0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
		130							57			28			

De acordo com a Teoria das Probabilidades, quando é utilizada a função modular, é bom que M seja um número primo. Sedgewick (2020) sugere escolher M da seguinte maneira: comece por escolher uma potência de dois que esteja próxima do valor desejado de M (ou seja, de um valor que seja apropriado para seus dados). Depois, adote para M o número primo que esteja logo abaixo da potência escolhida. A Figura 4 abaixo apresenta alguns exemplos.

Figura 4 – Criação de TH em função de números primos



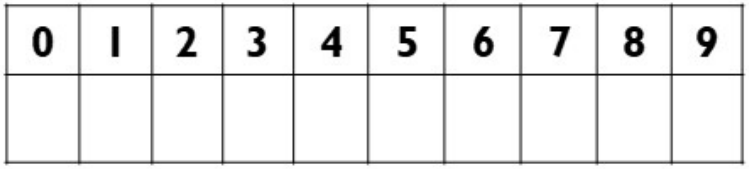
k	2^k	M
7	128	127
8	256	251
9	512	509
10	1024	1021
11	2048	2039
12	4096	4093
13	8192	8191
14	16384	16381
15	32768	32749
16	65536	65521
17	131072	131071
18	262144	262139

Fonte: Sidgewick (2020).

BINNING

Considere que você queira armazenar os valores de 0 a 999 em uma TH de tamanho 10. Uma função *hash* possível seria dividir o valor de entrada por 100 (modular). Assim, todas as chaves na faixa de 0 a 99 iriam para a entrada 0 da TH, as chaves de 100 a 199 iriam para a entrada 1, e assim por diante.

Qual entrada da TH terá mais elementos?



MID SQUARE

Nesta função, para se determinar a posição da chave na TH, elevaremos ao quadrado o valor a ser armazenado e tomaremos duas posições do cálculo, como o resultado da função *hash* (por exemplo, a quarta e a quinta posições, partindo-se da direita).

- Qual será o resultado da função h(2345)?

Calculamos = 2345 x 2345 = 54**99**025 – a chave 2345 deve ser armazenada na posição 99 da TH.

- Qual será o resultado da função h(180)?

Calculamos = 180 x 180 = **32**400 – a chave 180 deve ser armazenada na posição 32 da TH.

- Qual será o resultado da função h(26)?

Calculamos = 26 x 26 = **00**676 – a chave 26 deve ser armazenada na posição 00 da TH (repare que tivemos de preencher com zeros à esquerda para chegarmos nas quarta e quinta posições).

Funções Hash para Strings

Existem diversos tipos de funções hash para chaves do tipo String. Uma das mais comuns é a associação do valor ASCII de cada caractere, transformando a chave em um número inteiro. Veja um exemplo:

K = nome de pessoa

F(K) = obter o valor ASCII dos dois primeiros caracteres do nome, multiplicando-os e retendo os dois dígitos menos significativos:

Nome	**Cálculo**	**f(K)**
ALANA (A=65, L=76)	65 x 76 = 4940	40
BEATRIZ (B=66, E=69)	66 x 69 = 4554	54
LAURA (L=76, A=65)	76 x 65 = 4940	40
SOPHIA (S=83, O=79)	83 x 79 = 6579	79

Repare que as chaves ALANA e LAURA geram o mesmo valor f(k). Por esse motivo, recomenda-se a utilização de todo o conteúdo da chave(K).

Uma outra forma seria transformar a chave (string) em um número inteiro, a partir de seus valores ASCII e seguir com o método modular.

Vejamos um exemplo: armazenaremos uma sequência de strings utilizando a soma dos valores ASCII dos caracteres do módulo 599. Vamos armazenar as seguintes strings:

{“abcdef”, “bcdefa”, “cdefab”, “defabc”}.

Os valores ASCII de **a**, **b**, **c**, **d**, **e** e **f** são respectivamente 97, 98, 99, 100, 101 e 102. Como as strings contêm os mesmos caracteres permutados, a soma sempre será 597. Isso significa que todas as strings serão armazenadas na mesma posição da TH.

Tabela 2 – Função Hash para Strings e colisão

Índice				
0				
1				
2	abcdef	bcdefa	cdefab	defabc
3				
4				
...				

Adaptado de: Sidgewick (2020).

Aqui, será necessário um tempo O(n) (onde n é o número de strings) para buscar uma string específica. Isso mostra que a função hash não é boa.

Vamos tentar uma função hash diferente. O índice para uma string específica será igual à soma dos valores ASCII dos caracteres multiplicados por sua respectiva posição na string módulo 2069 (número primo).

Figura 5 – Função Hash para Strings sem colisão

String	Função Hash	Índice na TH
abcdef	(97**1 + 98**2 + 99**3 + 100**4 + 101**5 + 102**6) % 2069	38
bcdefa	(98**1 + 99**2 + 100**3 + 101**4 + 102**5 + 97**6) % 2069	23
cdefab	(99**1 + 100**2 + 101**3 + 102**4 + 97**5 + 98**6) % 2069	14
defabc	(100**1 + 101**2 + 102**3 + 97**4 + 98**5 + 99**6) % 2069	11

Adaptado de: Sidgewick (2020).

Lembre-se! Uma boa função hash deve...

- Minimizar as colisões.
- Ser rápida e fácil de calcular.
- Distribuir uniformemente os valores de chave na tabela *hash*.
- Usar todas as informações fornecidas na chave.

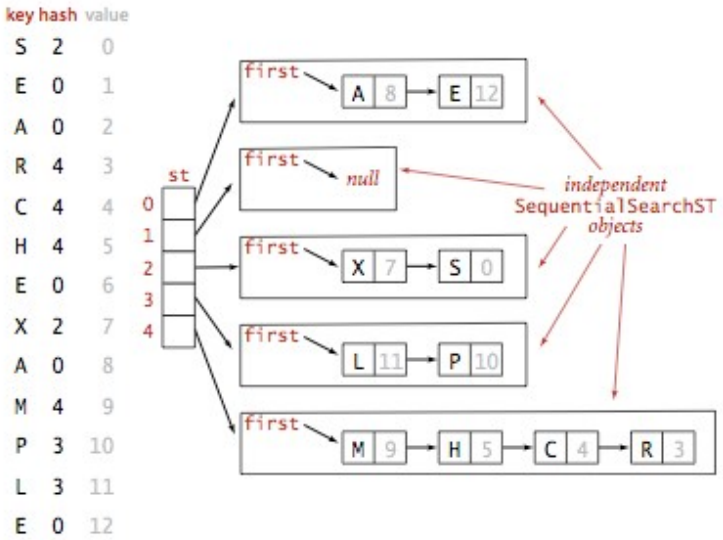
Tratando as Colisões

Uma função hash converte chaves em índices de TH. O segundo componente de um algoritmo de hash é a resolução de colisão: qual estratégia será usada quando duas ou mais chaves tiverem o mesmo valor de índice para a TH.

Separate Chaining

Uma das soluções é a construção, para cada um dos índices da TH, de uma lista ligada dos pares de valor-chave cujas chaves são indexadas a esse índice. A ideia básica é escolher M como suficientemente grande a fim de que as listas sejam suficientemente curtas para permitir a busca eficiente por meio de um processo em duas etapas: o hash para encontrar a lista que poderia conter a chave e a pesquisa sequencial por essa lista para encontrar a chave.

Figura 6 – Exemplo de Separate Chaining



Fonte: Sidgewick (2020).

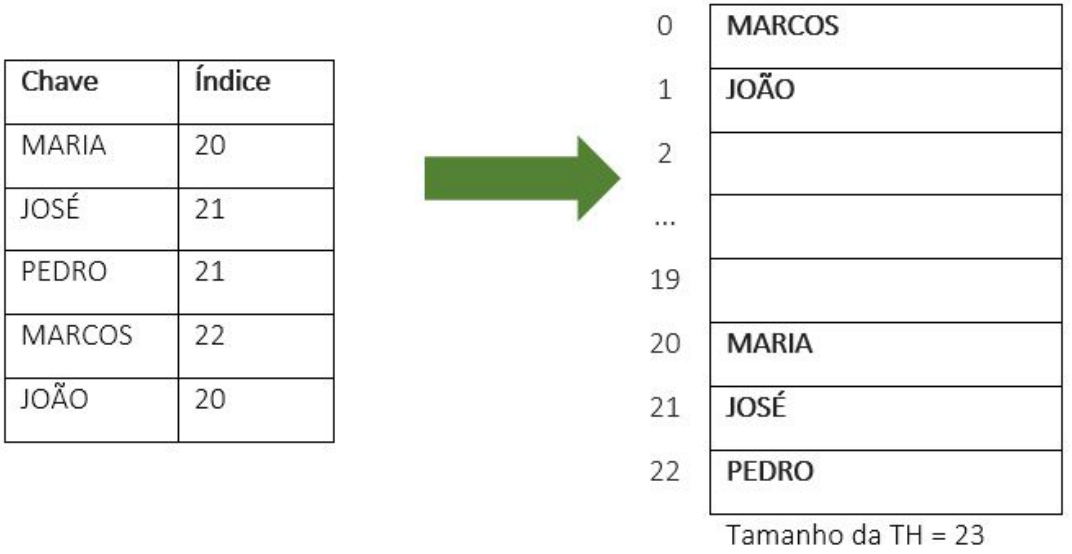
LINEAR PROBING

Outra abordagem para implementar hashing é armazenar N pares de valores-chave em uma tabela de hash de tamanho M > N, contando com entradas vazias na tabela para ajudar na resolução de colisões. Quando há uma colisão, então, apenas verificamos a próxima entrada na tabela (incrementando o índice).

Inserção de uma chave (K):

- Vá para a posição referente ao índice de k.
- Se estiver livre, armazene a chave.
- Se estiver ocupada, procure a primeira posição livre na sequência (atenção: a próxima posição da última posição da TH é a posição 0).

Figura 7 – Exemplo de Linear probing



Adaptado de: Sidgewick (2020).

Busca de uma chave (K):

- Vá para a posição referente ao índice de k.
- Se for o elemento procurado, fim da pesquisa.
- Caso contrário, tente a próxima posição até que a chave seja encontrada ou até que um espaço vazio seja identificado (elemento não existe).

Cálculo do tempo médio de busca

Como o tempo de busca aqui não é sempre constante, podemos calcular o tempo médio das buscas (soma da extensão da busca/ quantidade de elementos). Com base no exemplo anterior, veja a extensão de busca de cada um dos elementos, na tabela abaixo:

Tabela 3 – Tempo médio de busca

0	MARCOS
1	JOÃO
2	
...	
19	
20	MARIA
21	JOSÉ
22	PEDRO

Tamanho da TH = 23

Chave	Índices visitados
MARIA	1
JOSÉ	1
PEDRO	2
MARCOS	2
JOÃO	5
Tempo médio	11/5 = 2,2 buscas

Adaptado de: Sidgewick (2020).

REFERÊNCIAS

SEDGEWICK, Robert; WAYNE, Kevin. *Algorithms*, 2020. 4. ed. Disponível em: <https://algs4.cs.princeton.edu/home/>. Acesso em: 1º jul. 2021.