Fatec
Rio Claro

# Estrutura de Dados

Prof. Orlando Saraiva Júnior
orlando.nascimento@fatec.sp.gov.br

# Hash Tables

Fatec
Rio Claro

Uma tabela de hash é uma estrutura de dados que implementa um array associativo no qual os dados são armazenados mapeando as chaves para os valores como pares chave-valor.

Em muitas aplicações, geralmente precisamos de operações diferentes, como inserir, pesquisar e excluir, em uma estrutura de dados de dicionário.

# Hash Tables

# Hash Tables

Fatec
Rio Claro

Como sabemos, arrays e listas armazenam os elementos de dados em sequência. Assim como em uma array, os itens de dados são acessados por um número de índice.

Acessar elementos de uma matriz usando números de índice é rápido.

No entanto, eles são muito inconvenientes quando precisamos acessar qualquer elemento sem nos lembrarmos do número de índice. Por exemplo, se quisermos extrair o número de telefone de uma pessoa da agenda de endereços no índice 56, não há nada que vincule um contato específico ao número 56. É difícil recuperar uma entrada da lista usando o valor de índice.

# Hash Tables

Fatec
Rio Claro

Tabelas de hash são uma estrutura de dados mais adequada para esse tipo de problema. Uma tabela de hash é uma estrutura de dados em que os elementos são acessados por uma palavra-chave em vez de um número de índice, diferentemente de listas e matrizes. Nessa estrutura de dados, os itens de dados são armazenados em pares chave-valor, semelhantes a dicionários.

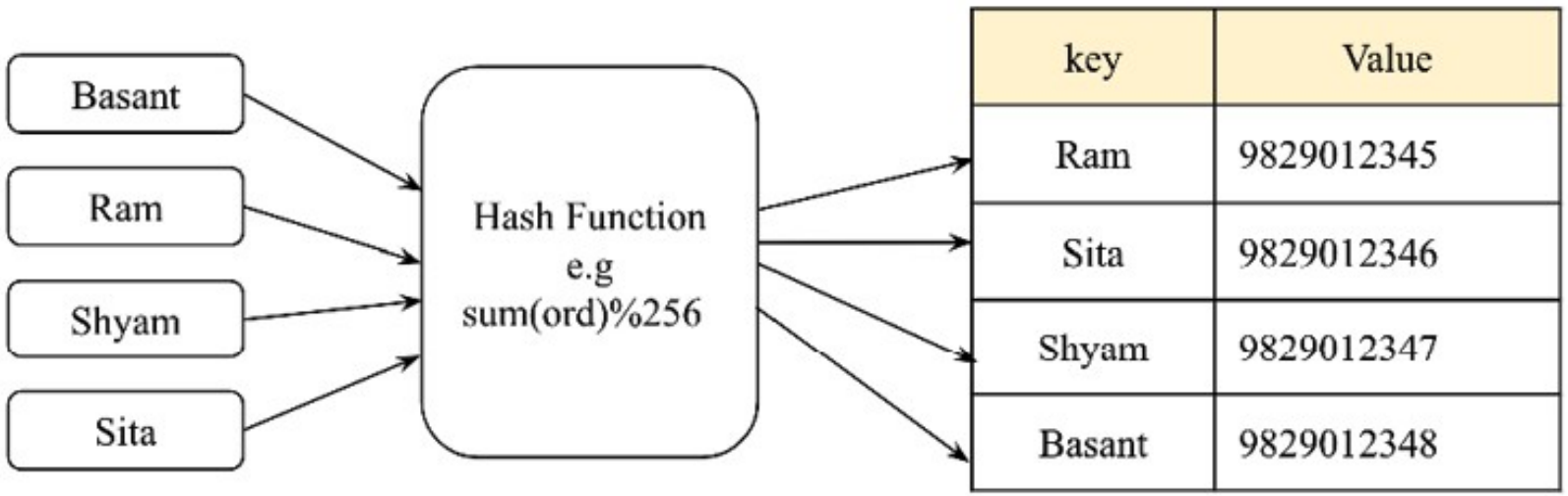
Uma tabela de hash usa uma função de hash para encontrar uma posição de índice onde um elemento deve ser armazenado e recuperado.

Isso nos proporciona consultas rápidas, pois estamos usando um número de índice que corresponde ao valor de hash da chave.

# Hash Tables

Fatec
Rio Claro

Uma visão geral de como a tabela de hash armazena os dados, na qual os valores-chave são transformados em hash usando qualquer função de hash para obter a posição de índice do registro na tabela de hash.



# Hash Tables

Fatec
Rio Claro

Dicionários são uma estrutura de dados amplamente utilizada, frequentemente construída usando tabelas de hash. Um dicionário usa uma palavra-chave em vez de um número de índice e armazena dados em pares (chave, valor).

Ou seja, em vez de acessar o contato com o valor do índice, usamos o valor da chave na estrutura de dados do dicionário.

# Prática

Fatec
Rio Claro

```
my_dict={"Basant" : "9829012345", "Ram": "9829012346",
"Shyam": "9829012347", "Sita": "9829012348"}


print("All keys and values")
for x,y in my_dict.items():
    print(x, ":" , y)
 my_dict["Ram"]
```

# Funções Hash

Fatec
Rio Claro

Hashing é uma técnica na qual, quando fornecemos dados de tamanho arbitrário a uma função, obtemos um valor pequeno e simplificado. Essa função é chamada de função hash.

O hashing usa uma função hash para mapear as chaves para outro intervalo de dados, de forma que um novo intervalo de chaves possa ser usado como índice na tabela hash; em outras palavras, o hashing é usado para converter os valores das chaves em valores inteiros, que podem ser usados como índice na tabela hash.

# Funções Hash

Fatec
Rio Claro

Em Python, a função ord() retorna um valor inteiro único (conhecido como valores ordinais) que é mapeado para um caractere no sistema de codificação Unicode.

Os valores ordinais mapeiam o caractere Unicode para uma representação numérica única, desde que o caractere seja compatível com Unicode.

Por exemplo, números de 0 a 127 são mapeados para caracteres ASCII, que também correspondem aos valores ordinais dentro dos sistemas Unicode.

No entanto, o intervalo de codificação Unicode pode ser maior.

# Funções Hash

Fatec Rio Claro

Além disso, para obter o hash da string inteira, poderíamos simplesmente somar os números ordinais de cada caractere da string. Veja o seguinte trecho de código:

```
sum(map(ord, 'hello world'))
```

Saída:

1116

# Funções Hash

Fatec Rio Claro

Na saída, obtemos um valor numérico, 1116, para a string hello world, que é o hash da string fornecida

| h | e | l | l | o | | w | o | r | l | d | |
|---|---|---|---|---|---|---|---|---|---|---|---|
| 104 | 101 | 108 | 108 | 111 | 32 | 119 | 111 | 114 | 108 | 100 | = 1116 |

# Funções Hash

Fatec
Rio Claro

A abordagem anterior, usada para obter o valor de hash de uma determinada string, apresenta o problema de que mais de uma string pode ter o mesmo valor de hash; por exemplo, quando alteramos a ordem dos caracteres na string e obtemos o mesmo valor de hash.

Veja o trecho de código a seguir, onde obtemos o mesmo valor de hash para a string "world hello":

```
sum(map(ord, 'world hello'))
```

# Funções Hash

Fatec Rio Claro

Novamente, haveria o mesmo valor de hash para a string "gello xorld", já que a soma dos valores ordinais dos caracteres dessa string seria a mesma.

```
sum(map(ord, 'gello xorld'))
```

| g | e | l | l | o | | x | o | r | l | d | |
|---|---|---|---|---|---|---|---|---|---|---|---|
| 103 | 101 | 108 | 108 | 111 | 32 | 120 | 111 | 114 | 108 | 100 | = 1116 |
| -1 | | | | | | +1 | | | | | |

# Funções Hash

Fatec
Rio Claro

Na prática, a maioria das funções de hash são imperfeitas e enfrentam colisões. Isso significa que uma função de hash atribui o mesmo valor de hash a mais de uma string. Essas colisões são indesejáveis para a implementação da tabela de hash.

# Funções Hash Perfeitas

Fatec
Rio Claro

Uma função de hash perfeita é aquela pela qual obtemos um valor de hash único para uma determinada string (pode ser qualquer tipo de dado; aqui, estamos usando um tipo de dado string como exemplo).

Nosso objetivo é criar uma função de hash que minimize o número de colisões, seja rápida, fácil de calcular e distribua os itens de dados igualmente na tabela de hash.

Mas, normalmente, criar uma função de hash eficiente que seja rápida e forneça um valor de hash único para cada string é muito difícil.

Se tentarmos desenvolver uma função de hash que evite colisões, isso se torna muito lento, e uma função de hash lenta não atende ao propósito da tabela de hash. Portanto, usamos uma função de hash rápida e tentamos encontrar uma estratégia para resolver as colisões em vez de tentar encontrar uma função de hash perfeita.

# Funções Hash Perfeitas

Fatec Rio Claro

Para evitar as colisões na função hash discutidas na seção anterior, podemos adicionar um multiplicador ao valor ordinal de cada caractere, que aumenta continuamente à medida que avançamos na string. Além disso, o valor hash da string pode ser obtido somando o valor ordinal multiplicado de cada caractere.

| h | e | l | l | o | | w | o | r | l | d | |
|---|---|---|---|---|---|---|---|---|---|---|---|
| 104 | 101 | 108 | 108 | 111 | 32 | 119 | 111 | 114 | 108 | 100 | = 1116 |
| 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 | 11 | |
| 104 | 202 | 324 | 432 | 555 | 192 | 833 | 888 | 1026 | 1080 | 1100 | = 6736 |

# Prática

Fatec
Rio Claro

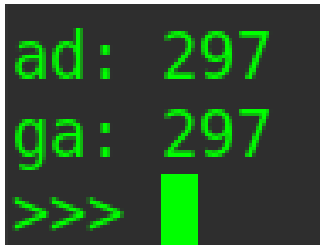
```
def myhash(s):
    mult = 1
    hv = 0
    for ch in s:
        hv += mult * ord(ch)
        mult += 1
    return hv
for item in ('hello world', 'world hello', 'gello xorld'):
    print("{}: {}".format(item, myhash(item)))
```

# Funções Hash Perfeitas

Fatec Rio Claro

Podemos ver que, desta vez, obtemos valores de hash diferentes para essas três strings. Ainda assim, este não é um hash perfeito. Vamos agora testar as strings **ad** e **ga**:

```
for item in ('ad', 'ga'):
    print("{}: {}".format(item, myhash(item)))
```



# Funções Hash Perfeitas

Fatec Rio Claro

Cada posição na tabela de hash é frequentemente chamada de *slot* ou *bucket* que pode armazenar um elemento.

Cada item de dados na forma de um par (chave, valor) é armazenado na tabela de hash em uma posição que é determinada pelo valor de hash da chave.

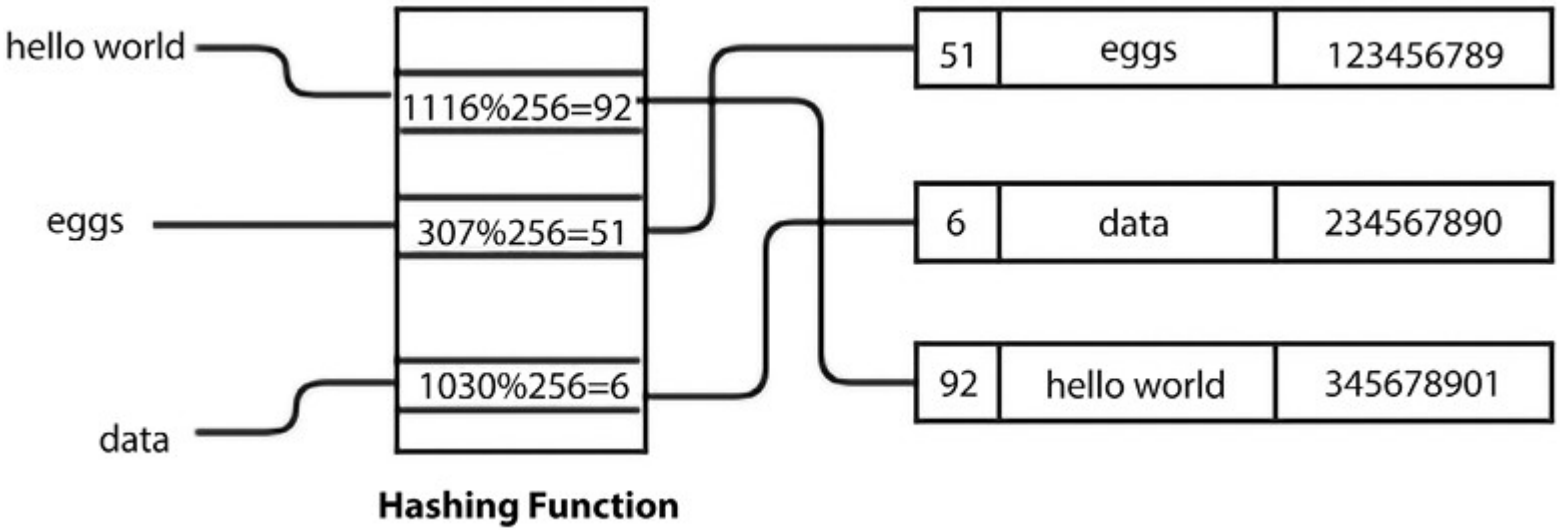
Vejamos um exemplo em que, primeiramente, usamos a função de hash que calcula o valor de hash somando os valores ordinais de todos os caracteres.

Em seguida, calculamos o valor de hash final (em outras palavras, a posição do índice) calculando os valores ordinais totais do módulo 256.

Aqui, usamos 256 slots/buckets como exemplo.

# Funções Hash Perfeitas

Fatec
Rio Claro



# Funções Hash Perfeitas

Fatec Rio Claro

Podemos usar qualquer número de slots, dependendo de quantos registros precisamos na tabela de hash.

Mostramos um hash de exemplo na anterior, que possui strings de chave correspondentes a valores de dados.

Por exemplo, a string de chave eggs possui o valor de dados correspondente 123456789.

Esta tabela de hash utiliza uma função de hash que mapeia a string de entrada *hello world* para um valor de *hash* de 92, que encontra uma posição de slot na tabela de hash.

# Funções Hash Perfeitas

Fatec
Rio Claro

Uma vez conhecido o valor de hash da chave, ele será usado para encontrar a posição onde o elemento deve ser armazenado na tabela de hash.

Portanto, precisamos encontrar um slot vazio. Começamos no slot que corresponde ao valor de hash da chave. Se esse slot estiver vazio, inserimos o item de dados nele.

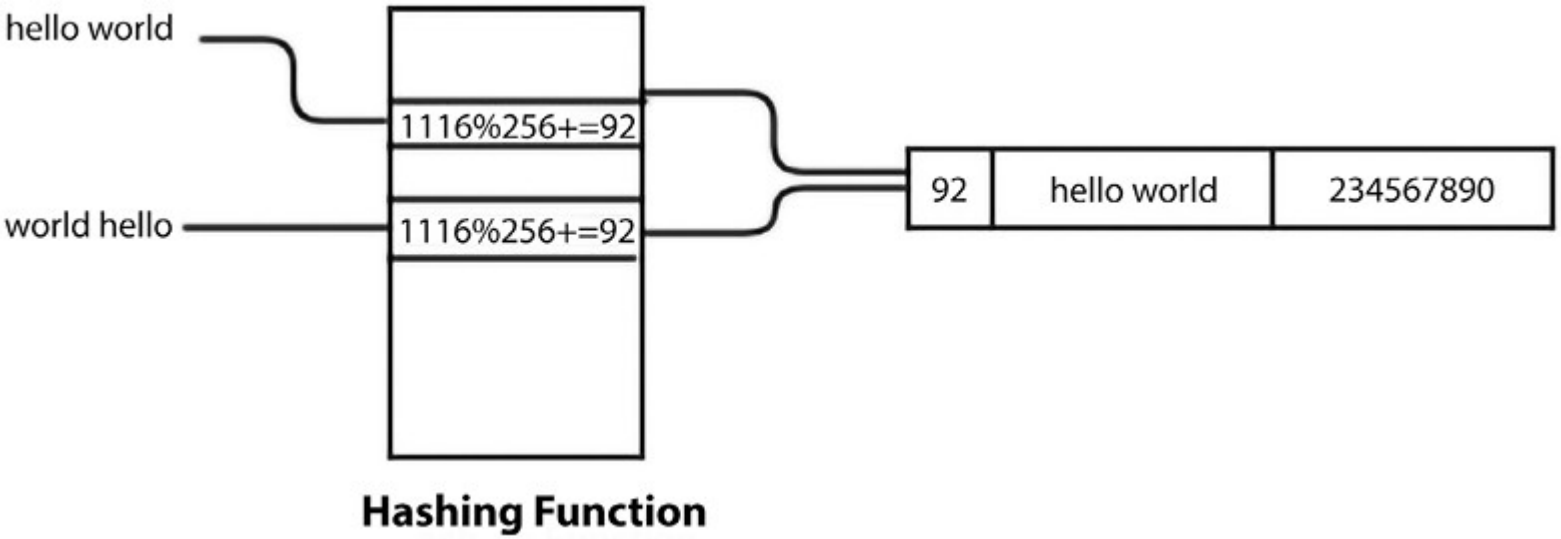
E, se o slot não estiver vazio, isso significa que temos uma colisão. Isso significa que temos um valor de hash para o item que é o mesmo de um item armazenado anteriormente na tabela.

Precisamos definir uma estratégia para evitar tais colisões ou conflitos.

# Funções Hash Perfeitas

Fatec Rio Claro

Por exemplo, a string de chave ***hello world*** já está armazenada na tabela na posição de índice 92 e, com uma nova string de chave, por exemplo, ***world hello***, obtemos o mesmo valor de hash de 92. Isso significa que há uma colisão.



# Funções Hash Perfeitas

Fatec Rio Claro

Uma maneira de resolver esse tipo de colisão é encontrar outro slot livre a partir da posição da colisão. Esse processo de resolução de colisão é chamado de endereçamento aberto (open addressing).

A colisão é resolvida por meio da busca (também chamada de sondagem) em uma posição alternativa até que se obtenha um slot não utilizado na tabela de hash para armazenar o item de dados.

Existem três abordagens populares para uma técnica de resolução de colisões baseada em endereçamento aberto:

- Sondagem linear
- Sondagem quadrática
- Hashing duplo

# Sondagem Linear

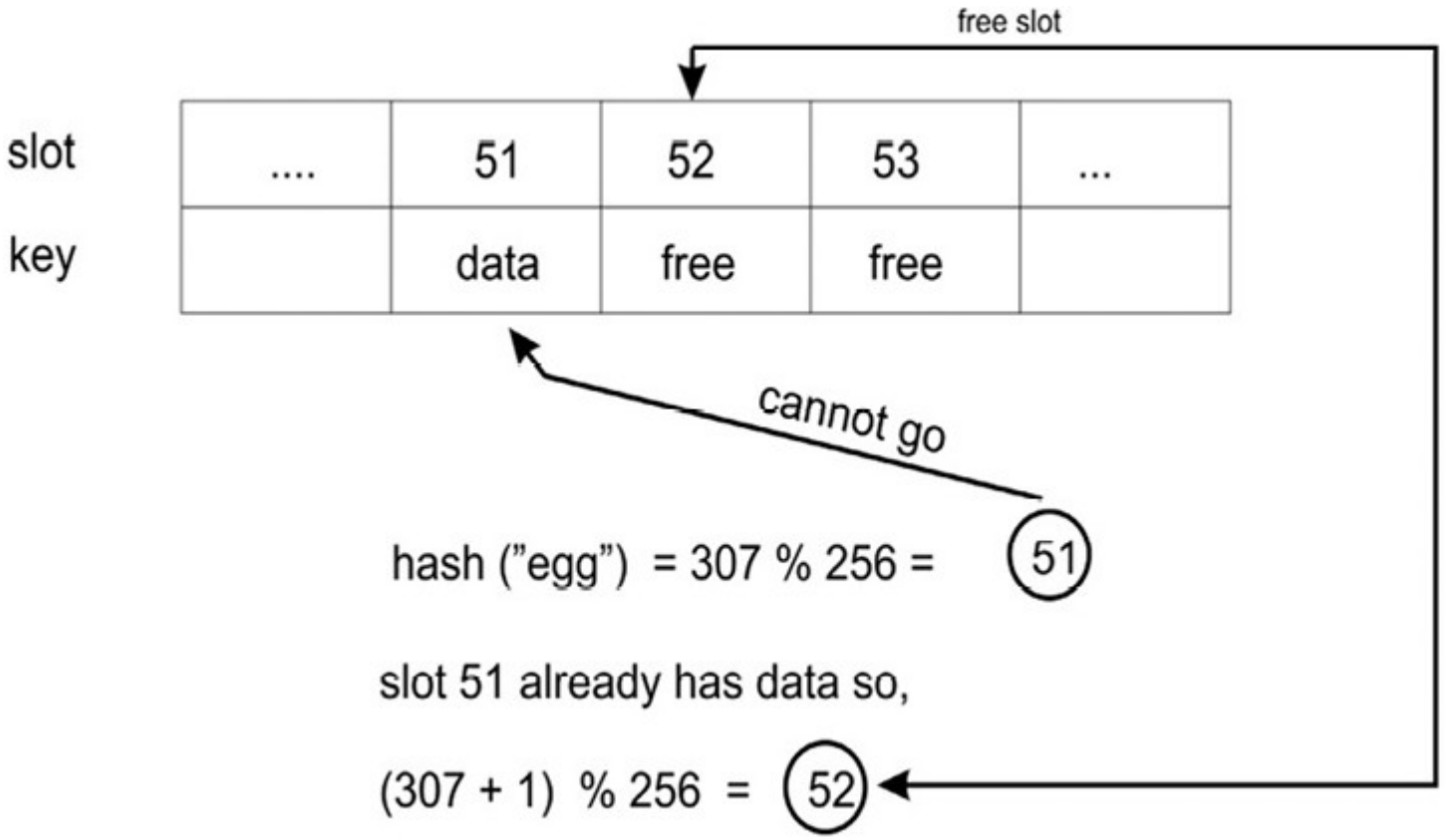
Fatec
Rio Claro

A maneira sistemática de visitar cada slot é uma forma linear de resolver colisões, na qual procuramos linearmente o próximo slot disponível adicionando 1 ao valor de hash anterior, onde obtemos a colisão.

Isso é conhecido como sondagem linear. Podemos resolver o conflito adicionando 1 à soma dos valores ordinais de cada caractere na string-chave, que é posteriormente usada para calcular o valor de hash final, considerando seu módulo de acordo com o tamanho da tabela de hash.

# Sondagem Linear

Fatec
Rio Claro



# Sondagem Linear

Fatec
Rio Claro

A maneira sistemática de visitar cada slot é uma forma linear de resolver colisões, na qual procuramos linearmente o próximo slot disponível adicionando 1 ao valor de hash anterior, onde obtemos a colisão.

Isso é conhecido como sondagem linear. Podemos resolver o conflito adicionando 1 à soma dos valores ordinais de cada caractere na string-chave, que é posteriormente usada para calcular o valor de hash final, considerando seu módulo de acordo com o tamanho da tabela de hash.

# Prática

Fatec
Rio Claro

Observe o código **hashtable.py**

# Sondagem Quadrática

Fatec
Rio Claro

Este também é um esquema de endereçamento aberto para resolver colisões em tabelas de hash. Ele resolve a colisão calculando o valor de hash da chave e adicionando valores sucessivos de um polinômio quadrático; o novo hash é computado iterativamente até que um slot vazio seja encontrado.

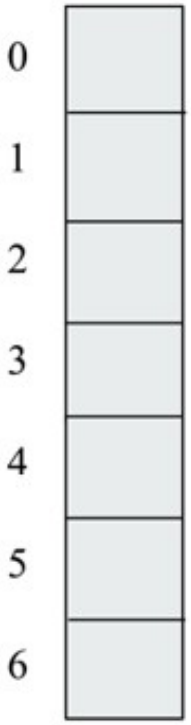
Se ocorrer uma colisão, os próximos slots livres são verificados nos locais $h + 1^2$, $h + 2^2$, $h + 3^2$, $h + 4^2$ e assim por diante.

Portanto, o novo valor de hash é calculado da seguinte forma:

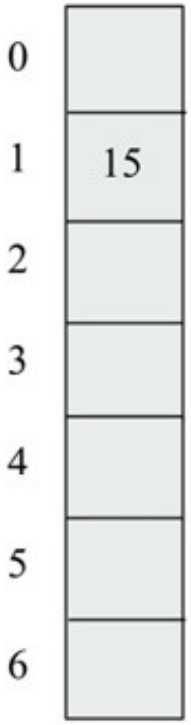
```
new-hash(key) = (old-hash-value + i²)
Here, hash-value = key mod table_size
```

# Sondagem Quadrática

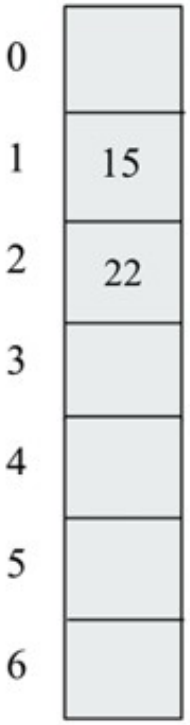
Fatec
Rio Claro



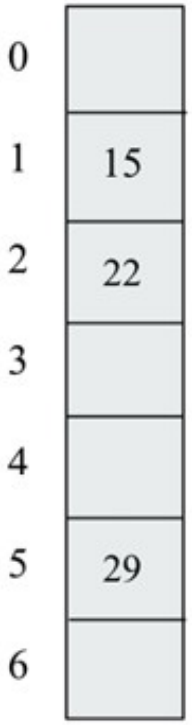

Empty table




Add element - 15
(15 mod 7)= 1




Add element - 22
(22 mod 7)= 1.
Collision here.
New position = (1 + $1^2$)




Add element-29, (29 mod 7)= 1.
Collision here.
New position = (1 + $1^2$).
Collision again.
New position = (1 + $2^2$) =5

# Hash Duplo

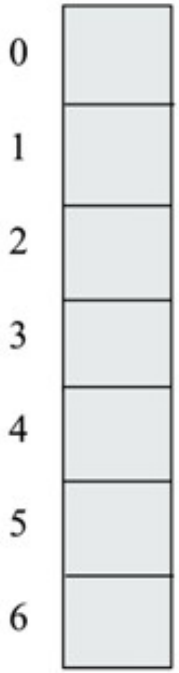
Fatec
Rio Claro

Na técnica de resolução de colisões por hashing duplo, usamos duas funções de hash. Essa técnica funciona da seguinte maneira. Primeiramente, a função de hash primária é usada para calcular a posição do índice na tabela de hash e, sempre que ocorre uma colisão, usamos outra função de hash para decidir o próximo slot livre para armazenar os dados, incrementando o valor de hash.

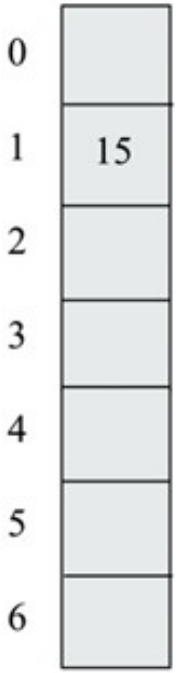
Para encontrar o próximo slot livre na tabela de hash, incrementamos o valor de hash, e esse incremento é fixo no caso de sondagem linear e sondagem quadrática. Devido a um incremento fixo no valor de hash quando ocorrem colisões, o registro é sempre movido para a próxima posição de índice disponível fornecida pela função de hash. Isso cria um cluster contínuo de posições de índice ocupadas.

# Hash Duplo

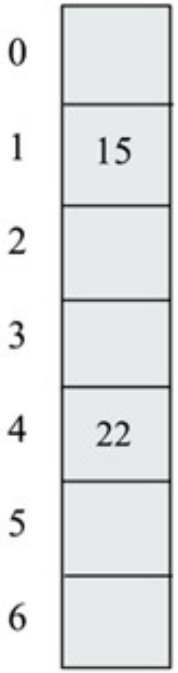
Fatec
Rio Claro



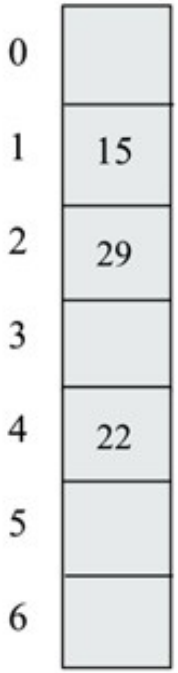

Empty table




Add element - 15
(15 mod 7)= 1




Add element - 22
(22 mod 7)= 1.
Collision here. New position
= (1+ 1*(5-(22 mod 5))) mod 7
= (1+3)mod 7
= 4




Add element-29, (29 mod 7)= 1.
Collision here. New position
= (1+ 1*(5-(29 mod 5))) mod 7
= (1+1) mod 7
= 2

# Encademamento Separado

# Encadeamento Separado

Fatec
Rio Claro

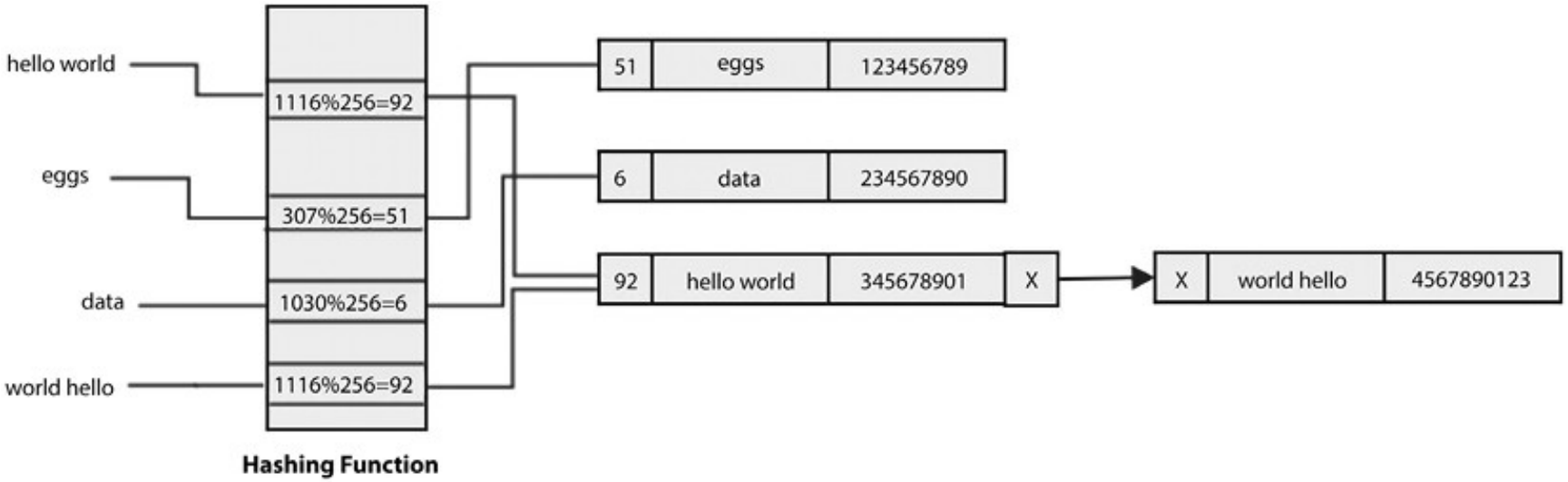
O encadeamento separado é outro método para lidar com o problema de colisão em tabelas de hash. Ele resolve esse problema permitindo que cada slot na tabela de hash armazene uma referência a vários itens na posição de uma colisão. Assim, no índice de uma colisão, podemos armazenar vários itens na tabela de hash.

No encadeamento, os slots na tabela de hash são inicializados com listas vazias. Quando um elemento de dados é inserido, ele é anexado à lista que corresponde ao valor de hash desse elemento.

No caso do encadeamento, ambos os elementos de dados são armazenados usando uma lista na posição de índice fornecida pela função de hash.

# Encadeamento Separado

Fatec
Rio Claro



# Encadeamento Separado

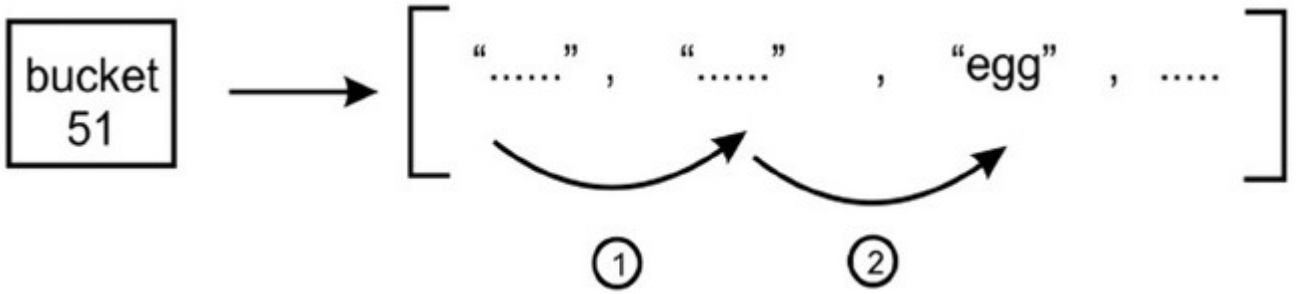
Fatec
Rio Claro

O encadeamento evita conflitos, permitindo que vários elementos tenham o mesmo valor de hash. Portanto, não há limite em termos do número de elementos que podem ser armazenados em uma tabela de hash, enquanto, no caso de técnicas de resolução de colisões por endereçamento aberto, tivemos que fixar o tamanho da tabela, que precisaríamos aumentar posteriormente quando a tabela estivesse preenchida. Além disso, a tabela de hash pode conter mais valores do que o número de slots disponíveis, já que cada slot contém uma lista que pode crescer.

No entanto, há um problema com o encadeamento: ele se torna ineficiente quando uma lista cresce em um determinado local de valor de hash. Precisamos fazer uma busca linear pela lista até encontrarmos o elemento que contém a chave desejada.

# Encadeamento Separado

Fatec Rio Claro

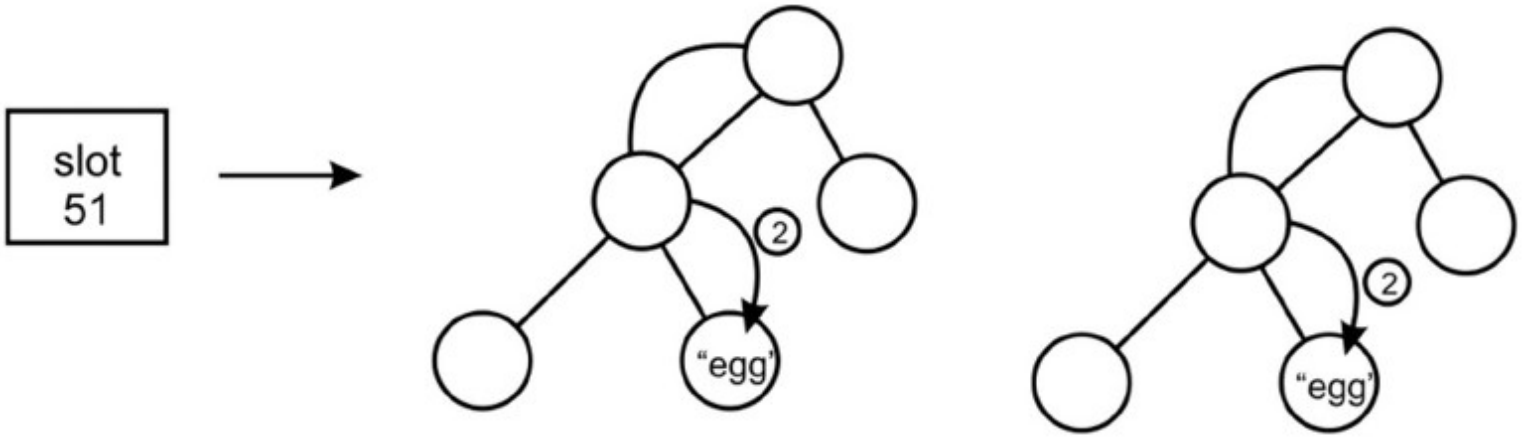



Há um problema com a recuperação lenta de itens quando uma posição específica em uma tabela hash possui muitas entradas.

Esse problema pode ser resolvido usando outra estrutura de dados em vez de uma lista que possa realizar buscas e recuperações rápidas. Há uma boa opção de usar árvores binárias de busca (BSTs),

# Encadeamento Separado

Fatec Rio Claro




O slot 51 contém um BST, que usamos para armazenar e recuperar os itens de dados.

Dependendo da ordem em que os itens fossem adicionados ao BST, poderíamos acabar com uma árvore de busca tão ineficiente quanto uma lista. Ou seja, cada nó na árvore tem exatamente um filho.

Para evitar isso, precisaríamos garantir que nosso BST seja autobalanceado.

# Tabela de Símbolos

# Tabela de Símbolos

Fatec
Rio Claro

Tabelas de símbolos são usadas por compiladores e interpretadores para rastrear os símbolos e diferentes entidades, como objetos, classes, variáveis e nomes de funções, que foram declarados em um programa. Tabelas de símbolos são frequentemente construídas usando tabelas de hash, pois é importante recuperar um símbolo da tabela com eficiência.

```
name = "Joe"
age = 27
```

# Tabela de Símbolos

Fatec
Rio Claro

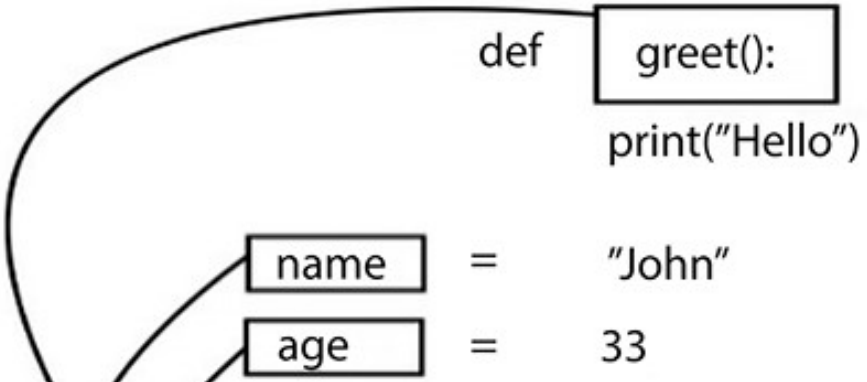
Cada símbolo possui um valor. Uma tabela de símbolos permite que o compilador ou o interpretador pesquise esses valores. Assim, os símbolos de nome e idade tornam-se chaves na tabela de hash. Todas as outras informações associadas a eles tornam-se o valor da entrada da tabela de símbolos.

O compilador cria uma tabela de símbolos para cada um de seus módulos, que são carregados na memória no momento de sua execução. As tabelas de símbolos são uma das aplicações importantes das tabelas de hash, que são usadas principalmente em compiladores e intérpretes para armazenar e recuperar eficientemente os símbolos e valores associados.

Em compiladores, as tabelas de símbolos também podem ter outros símbolos, como funções e nomes de classes. Por exemplo, a função greet() e duas variáveis, ou seja, nome e idade.

# Tabela de Símbolos

Fatec
Rio Claro




| symbol | type | value |
|---|---|---|
| greet | function | <function> |
| name | str | "John" |
| age | int | 33 |

# Outras aplicações

# Outras Aplicações

Fatec Rio Claro

## Armazenamento de Senhas (com Hashing Criptográfico)

Quando você cria uma conta e coloca uma senha, ela não fica salva em texto puro no banco de dados.

O sistema aplica uma função hash (como SHA-256, bcrypt, argon2) e guarda apenas o resultado.

Assim, mesmo que o banco seja invadido, não dá para ler diretamente sua senha.

## Verificação de Integridade de Arquivos (Checksums)

Quando você baixa um programa, muitas vezes aparece um “MD5” ou “SHA256”.

Isso é um hash do arquivo inteiro → se alguém alterar uma linha, o hash muda totalmente.

Serve para garantir que o arquivo não foi corrompido ou adulterado.

# Outras Aplicações

Fatec Rio Claro

## Índices em Bancos de Dados

Muitos bancos (como PostgreSQL, MongoDB, Redis) usam hash para acessar rapidamente registros.

Se você procurar um usuário pelo e-mail, o banco pode usar hash do e-mail como índice, em vez de varrer todos os registros.

## Caches e Sistemas de Armazenamento Rápido

Quando acessa uma página web, o servidor pode gerar um hash do conteúdo.

Esse hash serve como “chave” para guardar o conteúdo no cache → se pedir de novo, recupera mais rápido.

Exemplo: Memcached e Redis usam hash internamente.

# Outras Aplicações

Fatec Rio Claro

## Git e Controle de Versão

Cada commit do Git é identificado por um hash (SHA-1).

Isso garante que qualquer alteração no código muda completamente a “assinatura” do commit.

Assim, você tem integridade e histórico confiável do código.

## Sistemas de Arquivos (ex: EXT4, NTFS, ZFS)

Usam hash para organizar diretórios grandes.

Em vez de procurar um arquivo por nome percorrendo uma lista, usam hash do nome → acesso mais rápido.

# Síntese

# Síntese

Fatec
Rio Claro

Hoje discutimos técnicas de hash e a estrutura de dados de tabelas de hash. Aprendemos sobre a implementação e os conceitos de diferentes operações realizadas em tabelas de hash.

Também discutimos diversas técnicas de resolução de colisões, incluindo técnicas de endereçamento aberto, como sondagem linear, sondagem quadrática e hash duplo.

Por fim, analisamos tabelas de símbolos, que geralmente são construídas usando tabelas de hash. As tabelas de símbolos permitem que um compilador ou interpretador procure um símbolo (como uma variável, função ou classe) definido e recupere todas as informações sobre ele.

E comentamos sobre outras aplicações de hash

Fatec
Rio Claro

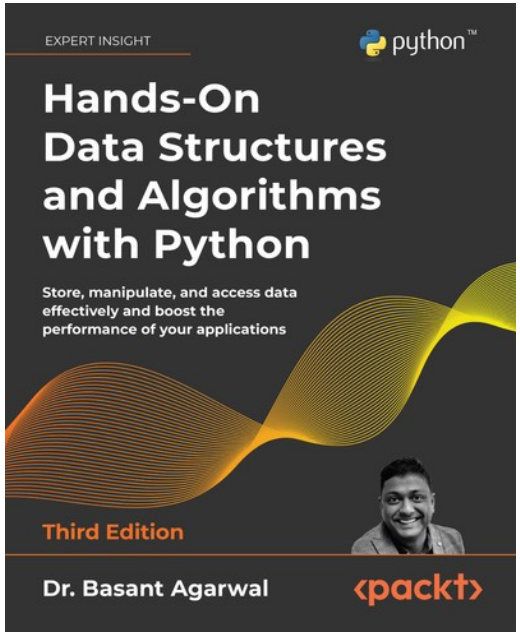
# Dúvidas

**Prof. Orlando Saraiva Júnior**
orlando.nascimento@fatec.sp.gov.br

# Prática

# Bibliografia

AGARWAL, Basant. Hands-On Data Structures and Algorithms with Python. 3. ed. Birmingham: Packt Publishing, 2022.



# Bibliografia

CANNING, John; BRODER, Alan; LAFORE, Robert. Data structures & algorithms in Python. Boston: Addison-Wesley Professional, 2019.