Capítulo 8: Tabela Hash

Uma tabela hash é uma estrutura de dados que implementa uma matriz associativa na qual os dados são armazenados mapeando as chaves para os valores como pares chave-valor. Em muitas aplicações, geralmente precisamos de diferentes operações, como inserção, pesquisa e exclusão em uma estrutura de dados de dicionário. Por exemplo, uma tabela de símbolos é uma estrutura de dados baseada em uma tabela hash que é usada pelo compilador. Um compilador que traduz uma linguagem de programação mantém uma tabela de símbolos na qual as chaves são cadeias de caracteres que são mapeadas para os identificadores. Em tais situações, uma tabela hash é uma estrutura de dados efetiva, pois podemos calcular diretamente o índice do registro necessário aplicando uma função de hash à chave. Assim, em vez de usar a chave como um índice de matriz diretamente, o índice da matriz é calculado aplicando a função de hash à chave. Isso torna muito rápido o acesso a um elemento a partir de qualquer índice da tabela hash. A tabela hash usa a função de hash para calcular o índice onde o item de dados deve ser armazenado na tabela hash.

Ao procurar um elemento na tabela hash, a hash da chave fornece o índice do registro correspondente na tabela. Idealmente, a função de hash atribui um valor único a cada uma das chaves; no entanto, na prática, podemos obter colisões de hash onde a função de hash gera o mesmo índice para mais de uma chave. Neste capítulo, discutiremos diferentes técnicas que lidam com essas colisões.

Neste capítulo, discutiremos todos os conceitos relacionados a esses, incluindo:
• Métodos de hash e técnicas de tabela hash
• Diferentes técnicas de resolução de colisões em tabelas hash

Introdução a Tabela Hash:

Como sabemos, arrays e listas armazenam os elementos de dados em sequência. Em um array, os itens de dados são acessados por um número de índice. O acesso aos elementos do array usando números de índice é rápido. No entanto, eles são muito inconvenientes de usar quando é necessário acessar qualquer elemento quando não podemos lembrar o número do índice. Por exemplo, se desejarmos extrair o número de telefone de uma pessoa do livro de endereços no índice 56, não há nada para vincular um contato específico com o número 56. É difícil recuperar uma entrada da lista usando o valor do índice.

As tabelas hash são uma estrutura de dados mais adequada para esse tipo de problema. Uma tabela hash é uma estrutura de dados em que os elementos são acessados por uma palavra-chave em vez de um número de índice, ao contrário de listas e arrays. Nesta estrutura de dados, os itens de dados são armazenados em

pares chave-valor, semelhantes a dicionários. Uma tabela hash usa uma função de hash para encontrar uma posição de índice onde um elemento deve ser armazenado e recuperado. Isso nos dá pesquisas rápidas, já que estamos usando um número de índice que corresponde ao valor de hash da chave.

Uma visão geral de como a tabela hash armazena os dados é mostrada na Figura 8.1, na qual os valores das chaves são hash usando qualquer função de hash para obter a posição de índice do registro na tabela hash.

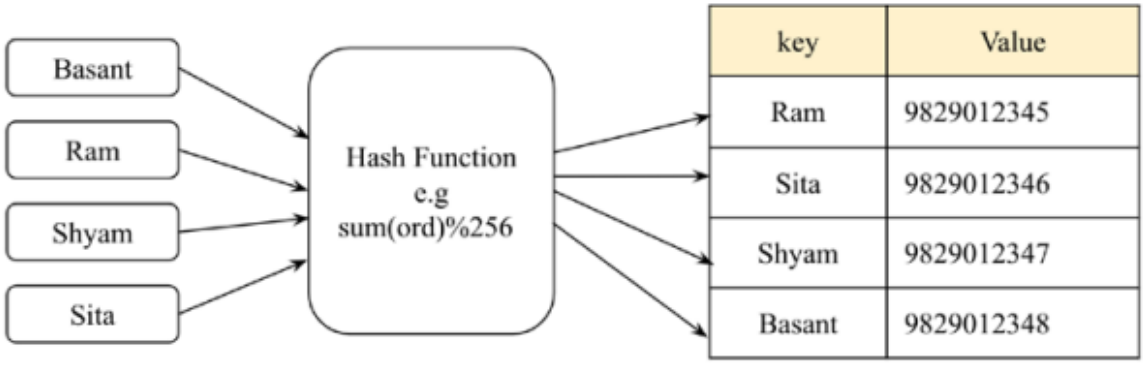

Figure 8.1: An example of a hash table

Dicionários são uma estrutura de dados amplamente utilizada, frequentemente construída usando tabelas 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. O seguinte código demonstra o funcionamento de dicionários que armazenam os dados em pares (chave, valor):

```
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) #prints keys and values
my_dict["Ram"]
```

SAÍDA:
Basant : 9829012345
Ram : 9829012346
Shyam : 9829012347
Sita : 9829012348

'9829012346'

As tabelas hash armazenam os dados de maneira muito eficiente para que a recuperação possa ser muito rápida. As tabelas hash são baseadas em um conceito chamado hashing.

Funções de Hash:

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 outra faixa de dados de tal forma que uma nova faixa de chaves possa ser usada como um 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.

Em nossas discussões neste capítulo, estamos usando hashing para converter strings em inteiros. Poderíamos ter usado qualquer outro tipo de dados em vez de strings que possam ser convertidos em inteiros. Vamos pegar um exemplo. Digamos que queremos fazer o hash da expressão "hello world", ou seja, queremos obter um valor numérico correspondente a essa string que possa ser usado como índice na tabela hash.

No 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 exclusiva, desde que o caractere seja compatível com Unicode, por exemplo, os números 0-127 são mapeados para caracteres ASCII, que também correspondem aos valores ordinais nos sistemas Unicode. No entanto, a faixa de codificação Unicode pode ser maior. Portanto, a codificação Unicode é um superconjunto do ASCII. Por exemplo, no Python, obtemos um valor ordinal exclusivo 102 para o caractere 'f' usando ord('f'). Além disso, para obter o hash de toda a string, podemos simplesmente somar os números ordinais de cada caractere na string. Veja o seguinte trecho de código:

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

SAÍDA:
1116

Na saída acima, obtemos um valor numérico, 1116, para a string "hello world", que é o hash da string fornecida. Considere a Figura 8.2 a seguir para ver os valores ordinais de cada caractere na string que resulta no valor hash 1116:

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

Figure 8.2: Ordinal values of each character for the hello world string

A abordagem anterior usada para obter o valor hash para uma determinada string tem o problema de que mais de uma string pode ter o mesmo valor hash; por exemplo, quando mudamos a ordem dos caracteres na string e temos o mesmo valor hash. Veja o seguinte trecho de código onde obtemos o mesmo valor hash para a string 'world hello':

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

SAÍDA:
1116

Novamente, haveria o mesmo valor hash para a string 'gello xorld', já que a soma dos valores ordinais dos caracteres para essa string seria a mesma, uma vez que 'g' tem um valor ordinal que é um menor do que o de 'h', e 'x' tem um valor ordinal que é um maior do que o de 'w'. Veja o seguinte trecho de código:

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

SAÍDA:
1116

Observe a Figura 8.3 a seguir, onde podemos ver que o valor hash para esta string 'gello xorld' é novamente 1116:

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

Figure 8.3: Ordinal values of each character for the gello xorld string

Na prática, a maioria das funções de hashing são imperfeitas e enfrentam colisões. Isso significa que uma função de hash dá o mesmo valor hash para mais de uma string. Tais colisões são indesejáveis para implementar a tabela de hash.

Perfeitas Funções Hash:

Uma função de hashing perfeita é aquela que gera um valor de hash único para uma determinada string (pode ser qualquer tipo de dado; aqui, estamos usando o tipo de dado string como exemplo). Nosso objetivo é criar uma função de hash que minimize o número de colisões, seja rápida, fácil de computar e distribua os itens de dados igualmente na tabela de hash. No entanto, normalmente, criar uma função de hash eficiente que seja rápida e forneça um valor de hash exclusivo 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 serve ao propósito da tabela de hash. Então, 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.

Para evitar colisões na função de hash discutida 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 de hash da string pode ser obtido adicionando o valor ordinal multiplicado de cada caractere. Para entender melhor o conceito, consulte a Figura 8.4 a seguir:

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

Figure 8.4: Ordinal values multiplied by numeric values for each character of the hello world string

Na Figura 8.4 anterior, o valor ordinal de cada caractere é progressivamente multiplicado por um número. Observe que a segunda linha tem os valores ordinais de cada caractere; a terceira linha mostra o valor do multiplicador; e, na quarta linha, obtemos valores multiplicando os valores das linhas dois e três, de modo que 104 x 1 seja igual a 104. Por fim, adicionamos todos esses valores multiplicados para obter o valor hash da string "hello world", que é 6736.

A implementação desse conceito é mostrada na seguinte função:

```
def myhash(s):
    mult = 1
    hv = 0
    for ch in s:
        hv += mult * ord(ch)
        mult += 1
    return hv
```

Podemos testar essa função nas strings que usamos anteriormente, mostradas a seguir:

```
for item in ('hello world', 'world hello', 'gello xorld'):
        print("{}: {}".format(item, myhash(item)))
```

When we execute the preceding code, we get the following output:

```
hello world: 6736
world hello: 6616
gello xorld: 6742
```

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

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

The output of the preceding code snippet is as follows:

```
ad: 297
ga: 297
```

Então, ainda não temos uma função de hash perfeita, pois obtemos os mesmos valores de hash para essas duas strings diferentes. Portanto, precisamos elaborar uma estratégia para resolver essas colisões. Discutiremos mais estratégias para resolver colisões nas próximas seções.

Resolvendo Colisões:

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 decidida pelo valor hash da chave. Vamos tomar um exemplo em que usamos a função de hash que calcula o valor hash somando os valores ordinais de todos os caracteres. Então, calculamos o valor hash final (ou seja, a posição do índice) calculando os valores ordinais totais de módulo 256. Aqui, usamos 256 slots/buckets como exemplo. Podemos usar qualquer número de slots, dependendo de quantos registros precisamos na tabela de hash. Mostramos um hash de amostra na Figura 8.5, que tem strings de chave correspondentes a valores de dados, por exemplo, a string de chave "eggs" tem o valor de dados correspondente a 123456789.

Esta tabela de hash usa 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:

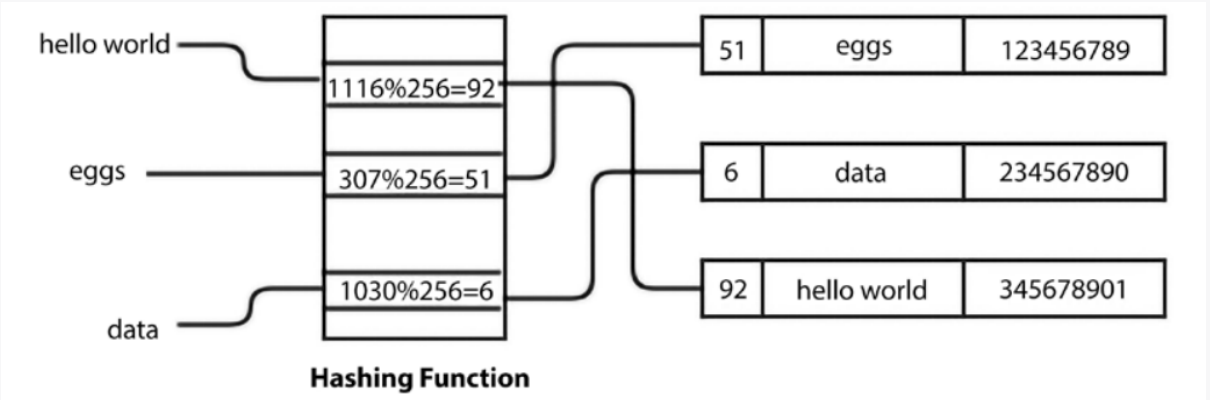

Figure 8.5: A sample hash table

Uma vez que conhecemos o valor 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 hash da chave. Se esse slot estiver vazio, inserimos o item de dados lá. E, se o slot não estiver vazio, significa que temos uma colisão. Isso significa que temos um valor hash para o item que é o mesmo que um item armazenado anteriormente na tabela. Precisamos determinar uma estratégia para evitar essas colisões ou conflitos.

Por exemplo, no diagrama a seguir, a string de chave "hello world" já está armazenada na tabela na posição do índice 92, e com uma nova string de chave, por exemplo, "world hello", obtemos o mesmo valor hash de 92. Isso significa que há uma colisão. Consulte a Figura 8.6 a seguir, que ilustra esse conceito:

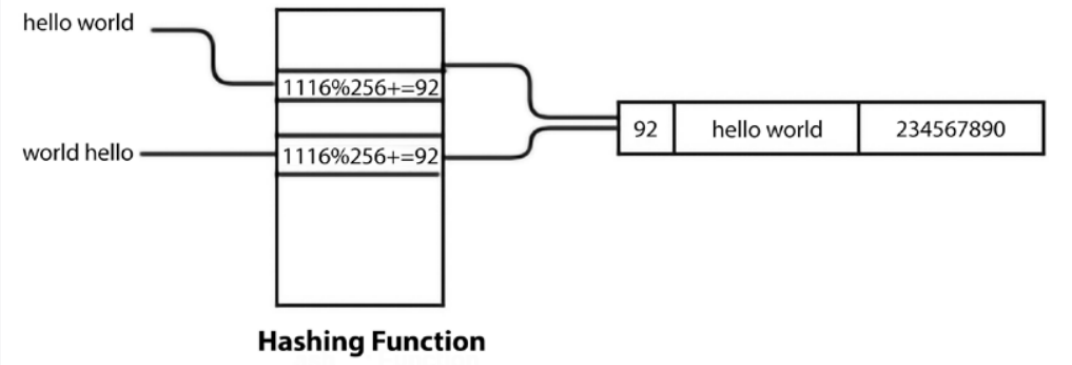

Figure 8.6: Hash values of two strings are the same

Uma maneira de resolver esse tipo de colisão é encontrar outro slot vazio a partir da posição da colisão. Esse processo de resolução de colisão é chamado de endereçamento aberto.

Endereço Aberto:

No endereçamento aberto, os valores de chave são armazenados na tabela hash e as colisões são resolvidas usando a técnica de sondagem. O endereçamento aberto é uma técnica de resolução de colisão usada em tabelas hash. A colisão é resolvida procurando (também chamado de sondagem) uma posição alternativa até obtermos um slot não utilizado na tabela hash para armazenar o item de dados.

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

1. Sondagem linear
2. Sondagem quadrática
3. Duplo hashing

Sondagem Linear:

A maneira sistemática de visitar cada slot é uma maneira linear de resolver colisões, na qual procuramos linearmente o próximo slot disponível adicionando 1 ao valor hash anterior onde ocorreu a colisão. Isso é conhecido como sondagem linear. Podemos resolver o conflito adicionando 1 à soma dos valores ordinais de cada caractere na string de chave, que é posteriormente usada para calcular o valor hash final, levando seu módulo de acordo com o tamanho da tabela hash.

Considere um exemplo. Primeiro, calculamos o valor hash da chave. Se a posição estiver ocupada, verificamos a tabela hash sequencialmente para o próximo slot livre. Vamos usar isso para resolver uma colisão, como mostrado na Figura 8.7 a seguir, onde, para a string de chave "egg", a soma dos valores ordinais chega a 307, e então calculamos o valor hash, levando o módulo 256, o que dá o valor hash para a string de chave "egg" como 51. No entanto, os dados já estão armazenados nesta posição, o que significa uma colisão. Portanto, adicionamos 1 ao valor hash que é calculado pela soma dos valores ordinais de cada caractere da string. Dessa forma, obtemos um novo valor hash, 52, para esta string de chave para armazenar os dados. Consulte a Figura 8.7 a seguir, que descreve o processo acima:

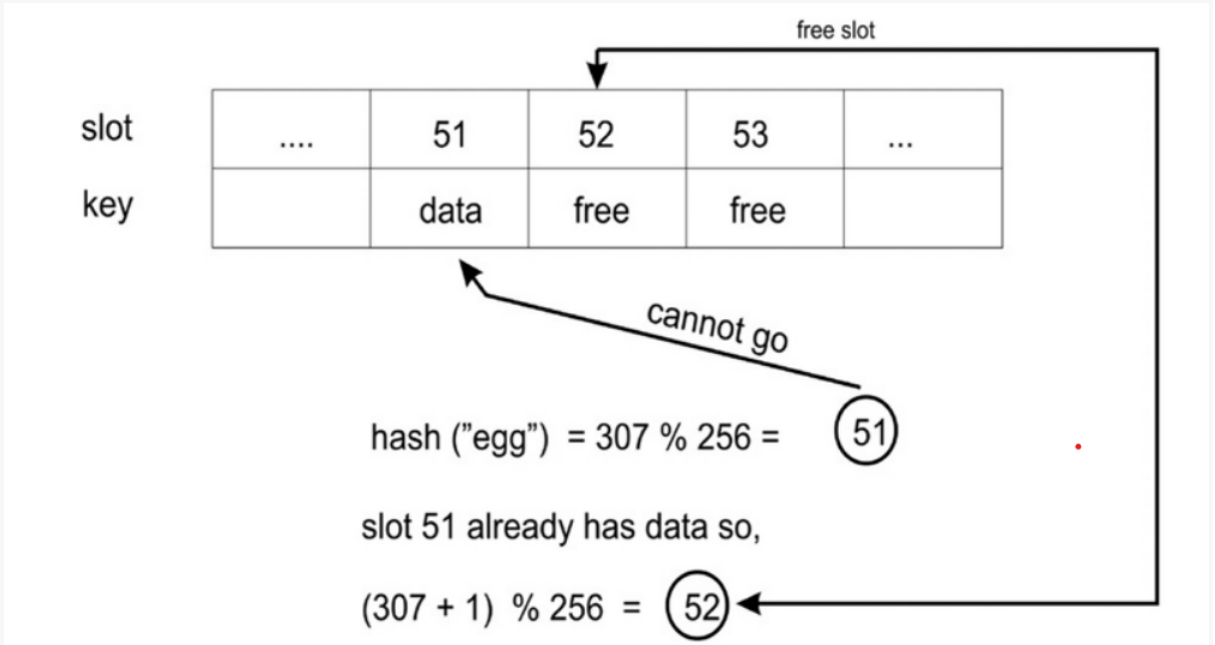

Figure 8.7: An example of collision resolution

Para encontrar o próximo espaço livre na tabela hash, incrementamos o valor de hash, e esse incremento é fixo no caso do linear probing. Devido ao incremento fixo no valor de hash quando há colisões, o novo elemento de dados é sempre armazenado na próxima posição de índice disponível fornecida pela função de hash. Isso cria um cluster contínuo de posições de índice ocupadas, com esse cluster crescendo sempre que obtemos outro elemento de dados que tem um valor de hash em qualquer lugar dentro do cluster. Portanto, uma grande desvantagem desse método é que a tabela hash pode ter posições consecutivas ocupadas que são chamadas de clusters de itens. Nesse caso, uma parte da tabela hash pode se tornar densa, enquanto a outra parte da tabela permanece vazia. Devido a essas limitações, podemos preferir usar uma estratégia diferente para resolver colisões, como quadrant probing ou double hashing, que discutiremos em seções futuras. Vamos primeiro discutir a implementação da tabela hash com linear probing como técnica de resolução de colisão e, após entender o conceito de linear probing, discutiremos outras técnicas de resolução de colisão.