Tabela Hash: cont. pag 256 até 268.

Implementando Tabelas Hash:

Para implementar a tabela hash, começamos criando uma classe para conter os itens da tabela hash. Eles precisam ter uma chave e um valor, já que a tabela hash é uma loja {chave-valor}:

```
class HashItem:
    def __init__(self, key, value):
        self.key = key
        self.value = value
```

A seguir, começamos a trabalhar na classe da tabela hash em si. Como de costume, começamos com um construtor:

```
class HashTable:
    def __init__(self):
        self.size = 256
        self.slots = [None for i in range(self.size)]
        self.count = 0
```

Listas padrão do Python podem ser usadas para armazenar elementos de dados em uma tabela hash. Vamos definir o tamanho da tabela hash para 256 elementos para começar. Mais tarde, vamos analisar estratégias de como aumentar a tabela hash à medida que começamos a preenchê-la. Agora, inicializamos uma lista contendo 256 elementos no código. Esses são as posições onde os elementos devem ser armazenados - os slots ou buckets. Então, temos 256 slots para armazenar elementos na tabela hash. É importante observar a diferença entre o tamanho e a contagem de uma tabela. O tamanho de uma tabela refere-se ao número total de slots na tabela (usados ou não). A contagem da tabela refere-se ao número de slots que estão preenchidos, ou seja, o número de pares reais (chave-valor) que foram adicionados à tabela.
Agora, precisamos decidir sobre uma função de hash para a tabela. Podemos usar qualquer função de hash. Vamos utilizar a mesma função de hash que retorna a soma dos valores ordinais de cada caractere nas strings com uma pequena modificação. Como esta tabela hash possui 256 slots, isso significa que precisamos de uma função de hash que retorne um valor no intervalo de 0 a 255 (o tamanho da tabela). Uma boa maneira de fazer isso é retornar o resto da divisão do valor de hash pelo tamanho da tabela, já que o resto certamente será um valor inteiro entre 0 e 255.

Como a função de hash destina-se apenas a ser usada internamente pela classe, colocamos um sublinhado (_) no início do nome para indicar isso. Esta é uma convenção do Python para indicar que algo é destinado ao uso interno. Aqui está a implementação da função de hash, que deve ser definida na classe HashTable:

```
def _hash(self, key):
    mult = 1
    hv = 0
    for ch in key:
        hv += mult * ord(ch)
        mult += 1
    return hv % self.size
```

Por enquanto, estamos assumindo que as chaves são strings. Discutiremos como chaves não-strings podem ser usadas posteriormente. Por enquanto, a função _hash() irá gerar o valor de hash para uma string.

Armazenando elementos em uma tabela hash:

Para armazenar os elementos na tabela hash, adicionamos eles à tabela com a função put() e os recuperamos com a função get(). Primeiramente, vamos ver a implementação da função put(). Começamos adicionando a chave e o valor à classe HashItem e depois computamos o valor de hash da chave. O método put() deve ser definido na classe HashTable:

```
def put(self, key, value):
    item = HashItem(key, value)
    h = self._hash(key)
    while self.slots[h] != None:
        if self.slots[h].key == key:
            break
        h = (h + 1) % self.size
    if self.slots[h] == None:
        self.count += 1
    self.slots[h] = item
    self.check_growth()
```

Depois de obter o valor de hash da chave e se o slot não estiver vazio, o próximo slot livre é verificado adicionando 1 ao valor de hash anterior, aplicando a técnica de linear probing. Considere o seguinte código:

```
while self.slots[h] != None:
    if self.slots[h].key == key:
        break
    h = (h + 1) % self.size
```

Se o slot estiver vazio, então aumentamos o count em um e armazenamos o novo elemento (o que significa que o slot anteriormente continha None) na lista na posição requerida. Consulte o seguinte código:

```
if self.slots[h] is None:
    self.count += 1
self.slots[h] = item
self.check_growth()
```

No código acima, criamos uma tabela hash e discutimos o método put() para armazenar o elemento de dados na tabela hash com a técnica de linear probing no momento da colisão.

Na última linha do código anterior, chamamos um método check_growth(), que é usado para expandir o tamanho da tabela hash quando temos um número muito limitado de slots vazios restantes na tabela hash. Vamos discutir isso com mais detalhes na próxima seção.

Expansão de uma tabela hash:

No exemplo que discutimos, fixamos o tamanho da tabela hash em 256. É óbvio que, quando adicionamos os elementos à tabela hash, ela começa a se encher e, em algum momento, todos os slots seriam preenchidos e a tabela hash estaria cheia. Para evitar tal situação, podemos aumentar o tamanho da tabela quando ela estiver começando a ficar cheia.

Para aumentar o tamanho da tabela hash, comparamos o tamanho e o count da tabela. size é o número total de slots e count denota o número de slots que contêm elementos. Portanto, se count for igual a size, isso significa que preenchemos a

tabela. O fator de carga da tabela hash é geralmente usado para expandir o tamanho da tabela; isso nos dá uma indicação de quantos slots disponíveis da tabela foram usados. O fator de carga da tabela hash é calculado dividindo o número de slots usados pelo número total de slots na tabela. É definido da seguinte forma:

Load factor (fator de carga) = n/k

Aqui, n é o número de slots usados e k é o número total de slots. À medida que o valor do fator de carga se aproxima de 1, isso significa que a tabela vai ser preenchida, e precisamos aumentar o tamanho da tabela. É melhor aumentar o tamanho da tabela antes que ela fique quase cheia, já que a recuperação de elementos da tabela fica lenta quando a tabela fica cheia. Um valor de 0,75 para o fator de carga pode ser um bom valor para aumentar o tamanho da tabela. Outra questão é o quanto devemos aumentar o tamanho da tabela. Uma estratégia seria simplesmente dobrar seu tamanho.

O problema do linear probing é que, à medida que o fator de carga aumenta, leva muito tempo para encontrar o ponto de inserção para o novo elemento. Além disso, no caso da técnica de resolução de colisão de endereçamento aberto, devemos aumentar o tamanho da tabela hash dependendo do fator de carga para reduzir o número de colisões.

A implementação do aumento da tabela hash quando o fator de carga aumenta mais do que o limite é a seguinte. Primeiro, redefinimos a classe HashTable que inclui mais uma variável, MAXLOADFACTOR, que é usada para garantir que o fator de carga da tabela hash esteja sempre abaixo do fator de carga máximo predefinido. A classe HashTable é definida da seguinte forma:

```
class HashTable:
    def __init__(self):
        self.size = 256
        self.slots = [None for i in range(self.size)]
        self.count = 0
        self.MAXLOADFACTOR = 0.65
```

A seguir, verificamos o fator de carga da tabela hash após adicionar qualquer registro à tabela hash usando o seguinte método check_growth(), que deve ser definido na classe HashTable:

```
def check_growth(self):
    loadfactor = self.count / self.size
    if loadfactor > self.MAXLOADFACTOR:
          print("Load factor before growing the hash table",
self.count / self.size)
        self.growth()
          print("Load factor after growing the hash table",
self.count / self.size)
```

No código anterior, calculamos o fator de carga da tabela e depois verificamos se é maior que o limite definido (ou seja, MAXLOADFACTOR é uma variável que inicializamos no momento da criação de uma tabela hash). Nesse caso, chamamos o método growth() que aumenta o tamanho da tabela hash (neste exemplo, estamos dobrando o tamanho da tabela hash). O método growth(), que deve ser definido na classe HashTable, é implementado da seguinte forma:

```
def growth(self):
    New_Hash_Table = HashTable()
    New_Hash_Table.size = 2 * self.size
    New_Hash_Table.slots = [None for i in range(New_Hash_Table.size)]

    for i in range(self.size):
        if self.slots[i] != None:
            New_Hash_Table.put(self.slots[i].key, self.slots[i].value)

    self.size = New_Hash_Table.size
    self.slots = New_Hash_Table.slots
```

No código anterior, primeiro criamos uma nova tabela hash com o dobro do tamanho da tabela hash original e, em seguida, inicializamos todos os seus slots como None. Em seguida, verificamos todos os slots preenchidos na tabela hash original, onde temos os dados, já que precisamos inserir todos esses registros existentes na nova tabela hash, portanto, chamamos o método put() com todos os pares de chave-valor da tabela hash existente. Depois de copiar todos os registros

para a nova tabela hash, substituímos o tamanho e os slots da tabela existente pela nova tabela hash.

Vamos criar uma tabela hash com capacidade máxima de 10 registros e um fator de carga limite de 65% definindo self.size = 10 no método init na classe HashTable, o que significa que sempre que um sétimo registro for adicionado à tabela hash, chamamos um método check_growth():

```
ht = HashTable()
ht.put("good", "eggs")
ht.put("better", "ham")
ht.put("best", "spam")
ht.put("ad", "do not")
ht.put("ga", "collide")
ht.put("awd", "do not")
ht.put("add", "do not")
ht.checkGrow()
```

No código acima, adicionamos sete registros usando o método put(). A saída do código anterior é a seguinte:

```
Load factor before growing the hash table 0.7
Load factor after growing the hash table 0.35
```

Na saída acima, podemos ver que o fator de carga antes e depois de adicionar o sétimo registro tornou-se a metade do fator de carga antes de expandir a tabela hash.

Na próxima seção, discutiremos o método get() para recuperar o elemento de dados que armazenamos na tabela hash.

Recuperando elementos da tabela hash

Para recuperar os elementos da tabela hash, o valor armazenado correspondente à chave seria retornado. Aqui, discutimos a implementação do método de recuperação - o método get(). Este método retorna o valor armazenado na tabela correspondente à chave fornecida.

Em primeiro lugar, calculamos o hash da chave dada correspondente ao valor que deve ser recuperado. Uma vez que temos o valor hash da chave, procuramos na tabela hash na posição do valor hash. Se o item da chave for correspondido com o valor da chave armazenada naquele local, o valor correspondente é recuperado.

Se isso não corresponder, então adicionamos 1 à soma dos valores ordinais de todos os caracteres na string, semelhante ao que fizemos no momento de armazenar os dados, e olhamos para o novo valor de hash obtido. Continuamos procurando até obtermos o elemento de chave ou verificamos todos os slots na tabela hash.

Aqui, usamos a técnica de sondagem linear para resolver a colisão e, portanto, usamos a mesma técnica ao recuperar o elemento de dados da tabela hash. Portanto, se usarmos uma técnica diferente, digamos, hash duplo ou sondagem quadrática no momento de armazenar o elemento de dados, devemos usar o mesmo método para recuperar o elemento de dados. Considere um exemplo para entender o conceito na Figura 8.8, e nos seguintes quatro passos:

1. Calculamos o valor hash para a string de chave dada, ovo, que acaba por ser 51. Em seguida, comparamos esta chave com o valor de chave armazenado no local 51, mas não corresponde.
2. Como a chave não corresponde, calculamos um novo valor hash.
3. Procuramos a chave no local do novo valor hash criado, que é 52; comparamos a string da chave com o valor da chave armazenado e, aqui, corresponde, como mostrado no seguinte diagrama.
4. O valor armazenado é retornado correspondente a este valor de chave na tabela hash. Veja a Figura 8.8 seguinte:

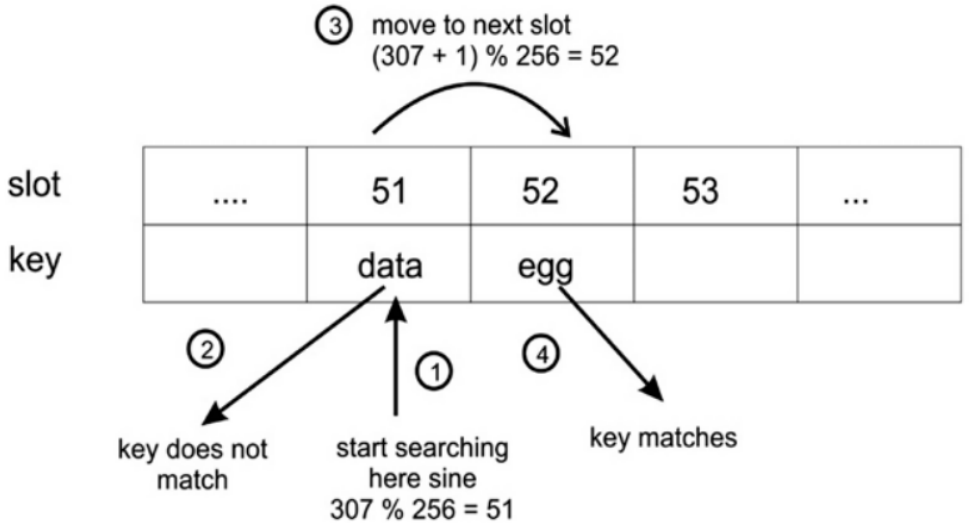

Figure 8.8: Four steps are demonstrated for retrieving an element from the hash table

Para implementar este método de recuperação, ou seja, o método get(), começamos calculando o hash da chave. Em seguida, procuramos o valor hash computado na tabela. Se houver uma correspondência, retornamos o valor armazenado correspondente. Caso contrário, continuamos procurando no novo local do valor hash calculado como descrito. Aqui está a implementação do método get(), que deve ser definido na classe HashTable:

```
def get(self, key):
    h = self._hash(key) # computed hash for the given key
    while self.slots[h] != None:
        if self.slots[h].key == key:
            return self.slots[h].value
        h = (h+ 1) % self.size
    return None
```

Finalmente, retornamos None se a chave não foi encontrada na tabela; poderíamos ter imprimido a mensagem de que a chave não foi encontrada na tabela de hash.

Testando a tabela hash

Para testar a tabela de hash, criamos uma HashTable e armazenamos alguns elementos nela e, em seguida, tentamos recuperá-los. Podemos usar o método get() para descobrir se um registro existe para uma determinada chave. Também usamos

as duas strings, "ad" e "ga", que tiveram a colisão e retornaram o mesmo valor de hash com nossa função de hash. Para avaliar o trabalho da tabela de hash, também incluímos essa colisão, apenas para ver se a colisão é resolvida adequadamente. Consulte o código de exemplo abaixo:

```
ht = HashTable()
ht.put("good", "eggs")
ht.put("better", "ham")
ht.put("best", "spam")
ht.put("ad", "do not")
ht.put("ga", "collide")

for key in ("good", "better", "best", "worst", "ad", "ga"):
    v = ht.get(key)
    print(v)
```

SAÍDA:

```
eggs
ham
spam
none
do not
collide
```

Como você pode ver, a busca pela chave "worst" retorna None, já que a chave não existe. As chaves "ad" e "ga" também retornam seus valores correspondentes, mostrando que a colisão entre elas é tratada adequadamente.

Implementando uma tabela hash como um dicionário:

Usar os métodos put() e get() para armazenar e recuperar elementos na tabela de hash pode parecer um pouco inconveniente. No entanto, também podemos usar a tabela de hash como um dicionário, pois seria mais fácil de usar. Por exemplo, gostaríamos de usar ht["good"] em vez de ht.get("good") para recuperar elementos da tabela.

Isso pode ser facilmente feito com os métodos especiais setitem() e getitem(), que devem ser definidos na classe HashTable. Veja o código a seguir:

```
def __setitem__(self, key, value):
    self.put(key, value)
def __getitem__(self, key):
    return self.get(key)
```

Agora, nosso código de teste ficaria assim:

```
ht = HashTable()
ht["good"] = "eggs"
ht["better"] = "ham"
ht["best"] = "spam"
ht["ad"] = "do not"
ht["ga"] = "collide"
for key in ("good", "better", "best", "worst", "ad", "ga"):
    v = ht[key]
    print(v)
print("The number of elements is: {}".format(ht.count))
```

A saída do código anterior é a seguinte:

```
eggs
ham
spam
none
do not
collide
The number of elements is: 5
```

Observe que também imprimimos o número de elementos já armazenados na tabela de hash usando a variável count. O código acima faz a mesma coisa que fizemos na seção anterior, mas é apenas mais conveniente de usar.

Na próxima seção, discutimos a técnica de sondagem quadrática para a resolução de colisões.

Quadratic probing:

Esta é também uma técnica de endereçamento aberto para resolver colisões em tabelas de hash. Ela resolve a colisão computando o valor de hash da chave e adicionando valores sucessivos de um polinômio quadrático; o novo valor de hash é iterativamente calculado até que um slot vazio seja encontrado. Se ocorrer uma colisão, os slots livres seguintes são verificados nas localizações $h + 1^2$, $h + 2^2$, $h + 3^2$, $h + 4^2$, 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
```

Quando temos uma chave como strings, calculamos o valor de hash usando a soma dos valores ordinais multiplicados por valores numéricos para cada caractere e, em seguida, passamos para a função de hash para finalmente obter o hash da chave string. No entanto, no caso de elementos de chave não string, podemos usar a função de hash diretamente para calcular o hash da chave.

Vamos tomar um exemplo simples de uma tabela hash na qual temos sete slots e assumimos que a função de hash é **h(key) = key mod 7**. Para entender o conceito de sondagem quadrática, vamos assumir que temos valores de elementos de chave que são o hash das strings de chave dadas.

Assim, sempre que usamos a técnica de sondagem quadrática para determinar as próximas posições de índice para armazenar um elemento de dados quando temos uma colisão, devemos seguir os seguintes passos para resolver a colisão:

1. Inicialmente, como temos uma tabela vazia, quando recebemos um elemento de chave de 15 (supondo que seja um hash da string dada), calculamos o

valor de hash usando nossa função de hash dada, em outras palavras, 15 mod 7 = 1. Assim, o elemento de dados é armazenado na posição de índice 1.

2. Em seguida, digamos que obtemos um elemento de chave de 22 (supondo que seja um hash da próxima string dada), usamos a função de hash para calcular o valor de hash, em outras palavras, 22 mod 7 = 1, ele dá a posição de índice 1. Como a posição de índice 1 já está ocupada, há uma colisão, então calculamos um novo valor de hash usando a sondagem quadrática, que é (1 + 1^2 = 2). A nova posição de índice é 2. Portanto, o elemento de dados é armazenado na posição de índice 2.
3. Em seguida, supondo que obtenhamos um elemento de dados de 29 (supondo que seja um hash da string dada), calculamos o valor de hash 29 mod 7 = 1. Como há uma colisão aqui, calculamos o valor de hash novamente como no passo 2, mas obtemos outra colisão aqui, então temos que calcular o valor de hash mais uma vez, em outras palavras (1 + 2^2 = 5), então os dados são armazenados naquela posição.

O exemplo acima de como resolver o processo usando a técnica de sondagem quadrática é mostrado na Figura 8.9:

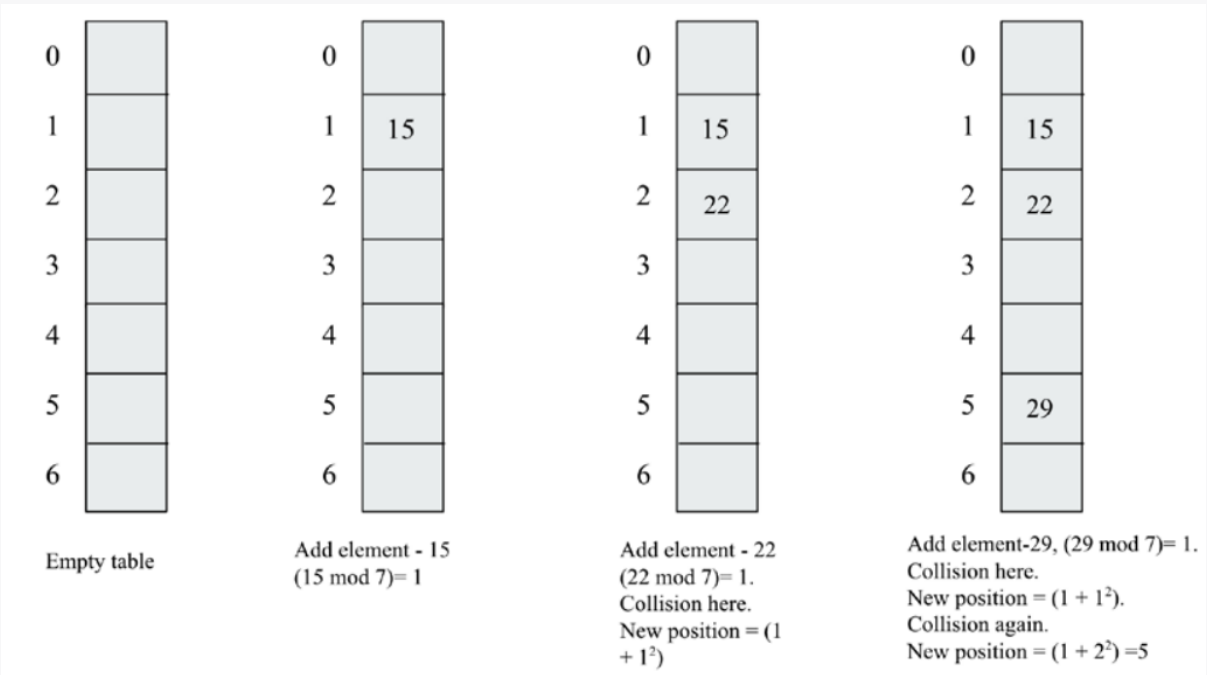

Figure 8.9: Example of collision resolution using quadratic probing

A técnica de sondagem quadrática para evitar colisões não sofre da formação de aglomerados de itens da mesma maneira que a sondagem linear; no entanto, sofre de agrupamento secundário. O agrupamento secundário cria uma longa sequência de slots preenchidos, uma vez que os elementos de dados que têm o mesmo valor de hash também terão a mesma sequência de sondagem.

Nós discutimos a implementação de uma tabela de hash na seção anterior com a adição e recuperação de elementos de dados, e usamos a técnica de sondagem linear para resolver a colisão. Agora, podemos atualizar a implementação da tabela de hash se quisermos usar qualquer outra técnica de resolução de colisão, como a técnica de sondagem quadrática. Todos os métodos serão os mesmos na classe HashTable, exceto os seguintes dois métodos, que devem ser definidos na classe HashTable:

```
def get_quadratic(self, key):
    h = self._hash(key)
    j = 1
    while self.slots[h] != None:
        if self.slots[h].key == key:
            return self.slots[h].value
        h = (h+ j*j) % self.size
        j = j + 1
    return None

def put_quadratic(self, key, value):
    item = HashItem(key, value)
    h = self._hash(key)
    j = 1
    while self.slots[h] != None:
        if self.slots[h].key == key:
            break
        h = (h + j*j) % self.size
        j = j+1
    if self.slots[h] == None:
        self.count += 1
    self.slots[h] = item
    self.check_growth()
```

O código acima dos métodos get_quadratic() e put_quadratic() é semelhante à implementação dos métodos get() e put() que discutimos anteriormente, exceto pelo fato de que as instruções de código estão em negrito nos códigos anteriores. As instruções em negrito indicam que, no momento da colisão, verificamos o próximo slot vazio usando a fórmula de sondagem quadrática:

```
ht = HashTable()
ht.put_quadratic("good", "eggs")
ht.put_quadratic("ad", "packt")
ht.put_quadratic("ga", "books")
v = ht.get_quadratic("ga")
print(v)
```

No código acima, primeiro adicionamos três elementos de dados juntamente com seus valores associados e, em seguida, procuramos um item de dados com a chave "ga" na tabela de hash.

A saída do código anterior é a seguinte:

```
books
```

A saída acima corresponde à string de chave "ga", que está correta de acordo com os dados de entrada armazenados na tabela de hash. Em seguida, discutiremos outra técnica de resolução de colisão - o hash duplo.

Double Hashing:

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

Para encontrar o próximo espaço livre na tabela hash, incrementamos o valor de hashing, e esse incremento é fixo no caso de sondagem linear e sondagem

quadrática. Devido ao incremento fixo no valor de hashing quando ocorrem colisões, o registro sempre é movido para a próxima posição de índice disponível dada pela função de hash. Isso cria um cluster contínuo de posições de índice ocupadas. Esse cluster cresce sempre que recebemos outro registro que tem um valor de hash em qualquer lugar dentro do cluster.

No entanto, no caso da técnica de double hashing, o intervalo de sondagem depende dos dados da chave em si, o que significa que sempre mapeamos para posições de índice diferentes na tabela hash sempre que ocorre uma colisão, o que, por sua vez, ajuda a evitar a formação de clusters.

A sequência de sondagem para essa técnica de resolução de colisão é a seguinte:

$(h^1(key) + i * h^2(key)) \bmod table\_size$
$h^1(key) = key \bmod table\_size$

É importante observar aqui que a segunda função de hash deve ser rápida, fácil de calcular, não deve avaliar para 0 e deve ser diferente da primeira função de hash.

Uma escolha para a segunda função de hash pode ser definida da seguinte forma:

$h^2(key) = prime\_number - (key \bmod prime\_number)$

Na função de hash acima, o número primo deve ser menor que o tamanho da tabela.

Por exemplo, digamos que tenhamos uma tabela hash que possa ter no máximo sete slots e que adicionemos os elementos de dados {15, 22, 29} a essa tabela em sequência. Os seguintes passos são realizados para armazenar esses elementos de dados na tabela hash usando a técnica de double hashing quando ocorre uma colisão:

1. Primeiramente, temos o elemento de dados 15, e calculamos o valor de hash usando a função de hash primária, em outras palavras, (15 mod 7 = 1). Como a tabela está vazia inicialmente, armazenamos os dados na posição do índice 1.
2. Em seguida, o elemento de dados é 22, e calculamos o valor de hash usando a função de hash primária, em outras palavras, (22 mod 7 = 1). Como a posição do índice 1 já está preenchida, isso significa que há uma colisão. Em

seguida, usamos a função de hash secundária definida acima como h2 (key) = número_primo - (key mod número_primo) para determinar as próximas posições de índice na tabela hash. Aqui, assumimos que o número primo menor que o tamanho da tabela é 5. Isso significa que a próxima posição de índice na tabela hash será (1 + 1 * (5 - (22 mod 5))) mod 7, o que equivale a 4. Então, armazenamos esse elemento de dados na posição do índice 4.

3. Em seguida, temos o elemento de dados 29, então calculamos o valor de hash usando a função de hash primária, ou seja, (29 mod 7 = 1). Obtemos uma colisão e agora usamos a função de hash secundária para estabelecer a próxima posição de índice para armazenar o elemento de dados, ou seja, (1 + 1*(5 - (29 mod 5))) mod 7, que acaba sendo 2, então armazenamos este elemento de dados na posição 2.