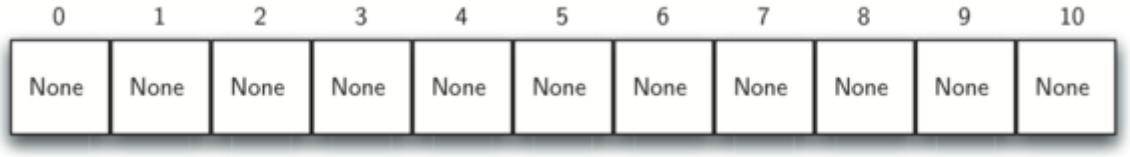
-> Hashing tem uma busca de o(1)
-> Adicionar e deletar através da tabela hash também é o(1)

Uma **tabela de dispersão** (ou hash table) é uma coleção de itens que são armazenados de maneira a serem encontrados com facilidade mais tarde. Cada posição da tabela de dispersão, geralmente denominada **índice** (ou slot), pode guardar um item e possui um rótulo inteiro começando a partir de 0
. Por exemplo, teremos um índice com rótulo 0, um índice com rótulo 1, outro com rótulo 2 e assim por diante. Inicialmente, a tabela de dispersão não contém nenhum item, então todos os índices estão vazios. Nós podemos implementar uma tabela de dispersão usando uma lista com cada elemento inicializado pelo valor especial `None` do Python. A Figura 4 mostra uma tabela de dispersão de tamanho

m=11

m=11. Em outras palavras, existem *m* índices na tabela, rotulados de 0 a 10.



O mapeamento entre a chave e o índice ao qual ela pertence na tabela é conhecido como a **função de espalhamento** (ou hash function). A função de espalhamento irá receber qualquer item na coleção e irá retornar um inteiro dentro do intervalo dos índices, isto é, entre 0 e *m*-1.

Nossa primeira função de hash, às vezes chamada de “método do resto”, simplesmente pega um item e o divide pelo tamanho da tabela, retornando o resto da divisão e o seu valor de espalhamento (

h(item)=item%11

h(item)=item%11). A Tabela 4 mostra todos os valores de espalhamento para os itens do nosso conjunto. Observe que o método do resto (artimética modular) tipicamente estará presente de algum modo em todas as funções de espalhamento, já que o resultado deve estar dentro do intervalo dos índices.

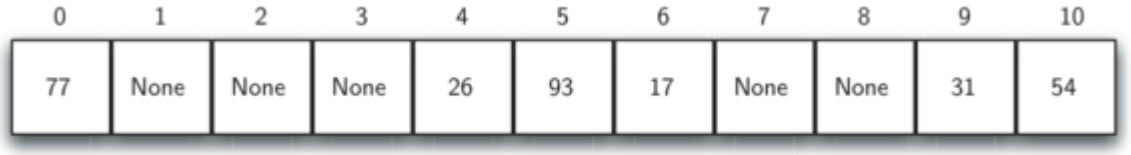
Item	Valor de Espalhamento
54	10
26	4

93	5
17	6
77	0
31	9

Uma vez que os valores de espalhamento tenham sido computados, podemos inserir cada item na tabela de dispersão na posição designada, como mostrado na Figura 5. Note que 6 dos 11 índices estão agora ocupados. Essa razão é conhecida como **fator de carga** e é comumente denotada por:

$\lambda =$ numerodeitens /tamanhodatabela

Para o nosso exemplo, o fator de carga é 6/ 11



Agora, quando queremos procurar um item, simplesmente utilizamos a função de espalhamento para calcular o índice correspondente da chave e acessamos a tabela de dispersão para verificar se ela está presente. Essa operação de busca é $O(1)$
Você provavelmente já deve ter notado que essa técnica só irá funcionar se cada item for mapeado uma localização única na tabela de dispersão. Por exemplo, se o item 44 fosse a próxima chave na nossa coleção, ele teria um valor de espalhamento de 0 ($44\%11==0$ 44%11==0). Como 77 também possui 0 como valor de espalhamento, teríamos um problema. Segundo a função de espalhamento, dois ou mais itens deveriam estar no mesmo índice. Isso é chamado de **colisão**. Claramente, colisões são um problema nesta técnica.

Funções de Espalhamento

Dada uma coleção de itens, uma função de espalhamento capaz de mapear cada item para um único índice é chamada de **função de espalhamento perfeita**. Se soubermos que os itens e a coleção nunca irá mudar, então é possível construir uma função de espalhamento perfeita. Infelizmente, dada uma coleção arbitrária de itens, não existe uma maneira sistemática de contruir uma função de espalhamento perfeita. Felizmente, não precisamos que a função de espalhamento seja pefeita para ainda termos ganhos de desempenho.
Uma forma de sempre termos uma função de espalhamento perfeita é aumentar o tamanho da tabela de dispersão, de modo que cada valor possível no intervalo de itens possa ser

acomodado. Isso garante que cada item terá um único índice. Embora isso seja algo prático para um número pequeno de itens, tal estratégia é inviável quando o número de itens é muito grande. Por exemplo, se os itens fossem os números de nove dígitos do INSS, esse método iria requerer quase um bilhão de índices. Se quiséssemos apenas armazenar os dados de uma sala de aula com 25 estudantes, estaríamos desperdiçando uma quantidade enorme de memória. Nosso objetivo é criar uma função de espalhamento que minimize o número de colisões, seja fácil de computar e distribua os itens uniformemente na tabela de dispersão. Há algumas maneiras conhecidas de estender o método do resto da divisão. Iremos considerar algumas delas agora.

O **método de folding** para construir funções de espalhamento começa dividindo o item em pedaços de tamanhos iguais (o último pedaço pode não ser de tamanho igual). Esses pedaços são somados para então gerar o valor de espalhamento resultante. Por exemplo, se o nosso item fosse o número de telefone 436-555-4601, iríamos extrair os dígitos e dividi-los em grupos de 2 (43, 65, 55, 46, 01). Depois de somar tudo,

$$43+65+55+46+01$$

43+65+55+46+01, teríamos 210. Se a nossa tabela de dispersão tiver 11 índices, então precisaríamos realizar o passo adicional de dividir o resultado por 11 e pegar o resto da divisão. Nesse caso,

$$210 \ \% \ 11$$

210 % 11 é 1, então o número de telefone 436-555-4601 seria mapeado para o índice 1. Alguns métodos de folding vão além e trocam a ordem dos pedaços antes de somá-los. Para o exemplo acima, teríamos algo como

$$43+56+55+64+01=219$$

43+56+55+64+01=219, o que resulta em

$$219 \ \% \ 11=10$$

219 % 11=10.

Outra técnica numérica para construir a função de espalhamento é conhecida como **método do quadrado do meio**. Nela, elevamos primeiro o item ao quadrado e depois extraímos uma parte dos dígitos resultantes. Por exemplo, se o item fosse 44, primeiro calculamos

$$44^{\,2}=1.936$$

442=1.936. Extraindo os dois dígitos do meio, 93, e realizando o passo de pegar o resto da divisão, ficamos com 5 ($93 \ \% \ 11$)

A Tabela 5 mostra valores de espalhamento calculados tanto para o método do resto da divisão quanto para o do quadrado do meio. Verifique se você entendeu como esses valores foram calculados.

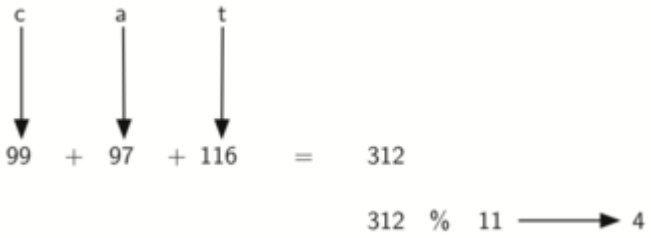
Item	Resto	Quadrado do Meio
54	10	3
26	4	7

93	5	9
17	6	8
77	0	4
31	9	6

Nós podemos criar também funções de espalhamento para itens baseados em caracteres, como strings. A palavra “gato” pode ser entendida como uma sequência de valores ordinais.

```
>>> ord('g')
103
>>> ord('a')
97
>>> ord('t')
116
>>> ord('o')
111
```

Podemos pegar esses quatro valores, somá-los e usar o método do resto da divisão para extrair um valor de espalhamento (veja a Figura 6). O Código 1 mostra uma função chamada `hash` que recebe uma string e o tamanho de uma tabela e retorna o valor de espalhamento dentro do intervalo de 0 a `tablesize`-1.



Código 1

```
def hash(astring, tablesize):
    sum = 0
    for pos in range(len(astring)):
        sum = sum + ord(astring[pos])

    return sum%tablesize
```

Você pode pensar em diversas outras formas de computar valores de espalhamento para os itens de uma coleção. O importante é lembrar que a função de espalhamento precisa ser eficiente, de modo que ela não se torne a parte dominante no processo de armazenamento e busca. Se a função de espalhamento for muito complexa, então computar posições na tabela se torna mais trabalhoso do que simplesmente realizar uma busca sequencial ou binária, já abordadas anteriormente. Isso iria aniquilar o propósito do hashing.

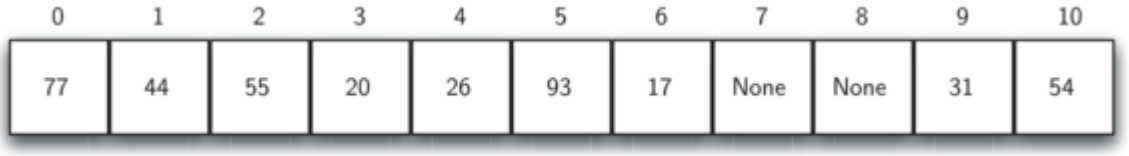
Resolução de Colisões

Retornamos agora ao problema das colisões. Quando dois itens são levados à mesma posição pela função de espalhamento, precisamos ter uma forma sistemática de colocar o segundo item na tabela de dispersão. Esse processo é chamado de **resolução de colisões**. Como dito anteriormente, se a função de espalhamento for perfeita, colisões nunca ocorrerão. Contudo, como isso não é muito realista, a resolução de colisões acaba sendo uma parte muito importante do hashing.

Um método para resolução de colisões olha para a tabela de dispersão e tenta encontrar outra posição aberta que possa armazenar o item que causou a colisão. Uma forma simples de fazer isso é começar pela posição do valor de espalhamento original e mover de forma sequencial pelas entradas até encontrar a primeira que esteja vazia. Note que poderemos ter que voltar para a primeira entrada (circularmente) para cobrir a tabela de dispersão inteira. Esse processo de resolução de colisões é conhecido como **endereçamento aberto**, já que ele procura encontrar a próxima entrada aberta na tabela de dispersão. Ao visitar sistematicamente uma posição por vez, estamos realizando uma técnica de endereçamento aberto chamada **sondagem linear**.

A Figura 8 mostra o conjunto ampliado de inteiros (54,26,93,17,77,31,44,55,20) submetidos à função de espalhamento simples baseada no resto da divisão. A Tabela 4 acima mostra os valores de espalhamento para os itens originais. A Figura 5 mostra seu elementos originais. Quando tentamos colocar o 44 na entrada 0, ocorre uma colisão. Usando a sondagem linear, procuramos sequencialmente, entrada por entrada, até encontrarmos uma posição aberta. Nesse caso, achamos o slot 1.

Novamente, o 55 deveria cair no slot 0, mas acaba sendo colocado no 2, já que é a próxima entrada disponível. O valor final de 20 leva à posição 9. Como ela já está ocupada, começamos a fazer a sondagem linear. Visitamos então as entradas 10, 0, 1 e 2, até finalmente encontrarmos uma entrada vazia na posição 3 da tabela.



Uma vez construída uma tabela de dispersão usando endereçamento aberto e sondagem linear, é essencial que utilizemos os mesmos métodos para procurar os itens. Suponha que estamos em busca do item 93. Quando computamos seu valor de espalhamento, temos 5 como resultado. Ao procurar na posição 5, encontramos o 93, então podemos retornar `True`. Mas o que acontece se procurarmos pelo 20? Nesse caso, o valor de espalhamento é 9 e o slot 9 está sendo ocupado pelo 31. Nós não podemos simplesmente retornar `False` porque sabemos que pode ter ocorrido colisões. Somos obrigados então a fazer uma busca sequencial, começando pela posição 10 e procurando até encontrarmos o item 20 ou uma entrada vazia.

Uma desvantagem da sondagem linear é a tendência para **aglutinação**, isto é, os itens tendem a ficar agrupados na tabela. Isso significa que se ocorrerem muitas colisões em um mesmo valor de espalhamento, as entradas vizinhas ficarão ocupadas por causa da sondagem linear. Isso terá um impacto sobre os outros itens que forem inseridos, como vimos quando tentamos adicionar o item 20 acima. Uma aglutinação de valores sendo levados a 0 teve que ser vencida para que finalmente encontrássemos uma posição vazia. Essa aglutinação é mostrada na Figura 9.

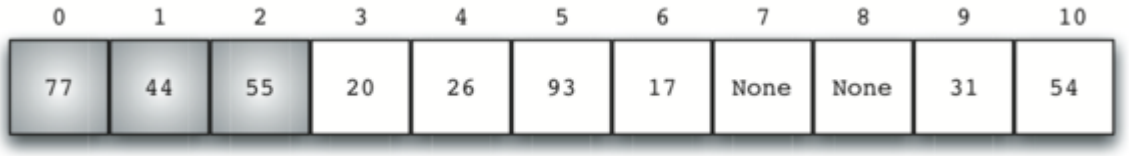
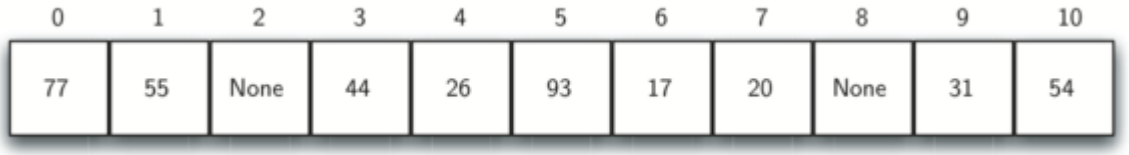


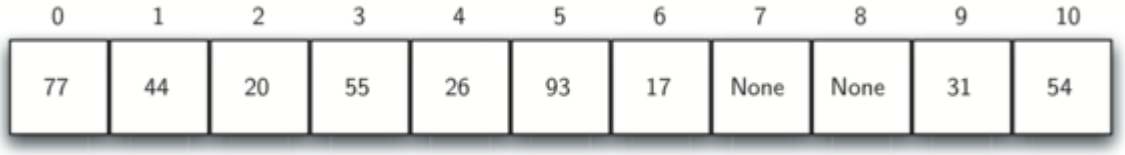
Figura 9: Uma Aglutinação de Items para a Entrada 0

Uma forma de lidar com a aglutinação é estender a técnica de sondagem linear para que em vez de procurar sequencialmente para a próxima entrada aberta, alguns slots sejam pulados, distribuindo assim os itens que causaram colisão de maneira mais uniforme. Isso potencialmente irá reduzir a aglutinação. A Figura 10 mostra como os itens ficam dispostos quando uma resolução por colisão é feita com uma sondagem “mais 3”. Isso significa que uma vez ocorrida a colisão, iremos procurar de três em três entradas até encontrar uma vazia

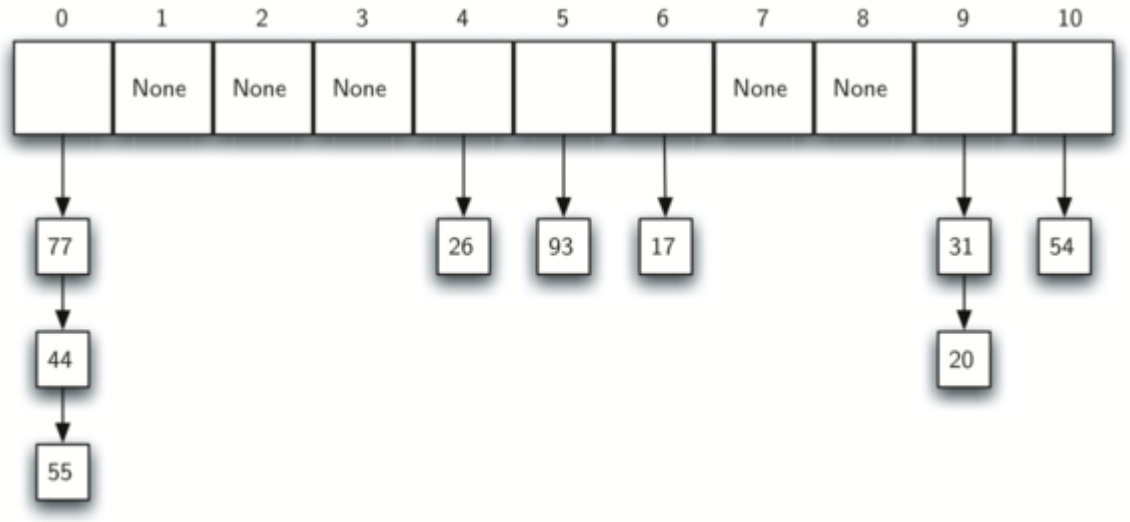


O nome geral para esse processo de procurar por outro slot depois de uma colisão é **rehashing**. Com uma simples sondagem linear, a função de rehash é
newhashvalue=rehash(oldhashvalue)
newhashvalue=rehash(oldhashvalue), onde
rehash(pos)=(pos+1)%tamanhodatabela
rehash(pos)=(pos+1)%tamanhodatabela. O rehash “mais 3” pode ser definido como
rehash(pos)=(pos+3)%tamanhodatabela
rehash(pos)=(pos+3)%tamanhodatabela. De modo geral, temos
rehash(pos)=(pos+pulo)%tamanhodatabela
rehash(pos)=(pos+pulo)%tamanhodatabela. É importante notar que o tamanho do “pulo” precisa assumir um valor tal que todas as entradas da tabela serão visitadas em algum momento. Caso contrário, parte da tabela será inutilizada. Para garantir isso, sugere-se que o tamanho da tabela seja um número primo. Essa é a razão de termos escolhido 11 em nossos exemplos.

Uma variação da ideia de sondagem linear é a **sondagem quadrática**. Em vez de um valor constante de “pulo”, usamos uma função de reshash que incrementa o valor espalhamento por 1, 3, 5, 7, 9 e assim por diante. Isso significa que se o primeiro valor de espalhamento for *h*, os valores seguintes são
h+1
h+1,
h+4
h+4,
h+9
h+9,
h+16
h+16 e assim sucessivamente. Em outras palavras, a sondagem quadrática utiliza um pulo que consiste de quadrados perfeitos sucessivos. A Figura 11 mostra nossos valores de exemplo depois de eles terem sido alocados usando essa técnica.



Um método alternativo para lidar com o problema da colisão é permitir que cada entrada tenha uma referência para uma coleção (ou sequência) de itens. O **Encadeamento** permite que muitos itens existam no mesma entrada de uma tabela de dispersão. Quando colisões acontecem, o item ainda assim é colocado na entrada apropriada da tabela de dispersão. Mas conforme mais itens vão sendo alocados para a mesma posição, a dificuldade de encontrar um item na coleção aumenta. A Figura 12 mostra os itens conforme eles aumentb. vão sendo adicionados a uma tabela de dispersão que usa encadeamento aumentc. para resolver colisões.



Quando queremos procurar por um item, usamos a função de espalhamento para gerar o slot aonde ele deveria cair. Como cada entrada contém uma coleção, usamos uma técnica de busca para decidir se o item está presente ou não. A vantagem é que na média é provável que haja muito poucos itens por slot, então a busca tende a ser mais eficiente.

Um dos tipos mais úteis de coleção em Python é o dicionário. Lembre-se de que um dicionário é tipo de dado associativo onde você poder armazenar pares de chave-valor. A chave é usada para procurar pelo valor associado. Com frequência, referimo-nos a essa ideia como um **mapa**.

O tipo de dado abstrato mapa é definido da seguinte forma: a estrutura é uma coleção não ordenada de associações entre uma chave e um valor. As chaves em um mapa são todas únicas de modo que há uma correspondência um-para-um entre uma chave e um valor. As operações são dadas a abaixo.

- `Map()` Cria um novo mapa vazio. Retorna uma coleção do tipo mapa vazia.
- `put(key,val)` Adiciona um novo par chave-valor para o mapa. Se a chave já estiver no mapa, então substitui o valor antigo pelo novo.
- `get(key)` Dada uma chave, retorna o valor armazenado no mapa ou `None`, caso contrário.
- `del` Elimina o par chave-valor do mapa usando uma declaração da forma `del map[key]`.
- `len()` Retorna o número de pares chave-valor armazenadas no mapa.
- `in` Retorna `True` para uma declaração da forma `key in map`, se a chave dada estiver no mapa e `False`, caso contrário.

Um dos grandes benefícios de um dicionário é o fato de que dada uma chave, podemos procurar pelo valor associado rapidamente. Para conseguir realizar essa busca rápida, precisamos de uma implementação que dê suporte a uma busca eficiente. Poderíamos usar

uma lista com busca binária ou sequencial, mas seria ainda melhor se usássemos uma tabela de dispersão, como descrita acima, já que a procura por um item em uma tabela de dispersão pode ter um desempenho de aproximadamente
O(1)
Em Listing 2, usamos duas listas para criar uma classe do tipo `HashTable` que implementa o tipo de dado abstrato Map. Uma lista, chamada `slots`, irá armazenar os valores. Quando realizamos a busca com uma chave, a posição correspondente na lista de dados irá conter o valor associado. Iremos tratar a lista de chaves como uma tabela de dispersão, de acordo com as ideias apresentadas anteriormente. Note que o tamanho inicial escolhido para a tabela de dispersão foi 11. Embora isso seja arbitrário, é importante que o tamanho seja um número primo para que o algoritmo de resolução de colisões seja o mais eficiente possível.

Listing 2

```
class HashTable:
    def __init__(self):
        self.size = 11
        self.slots = [None] * self.size
        self.data = [None] * self.size
```

A função `hashfunction` é uma simples implementação do método do resto. A ténica para resolução de colisões é a sondagem linear com uma função de rehash “mais 1”. A função `put` (veja Listing 3) pressupõe que em algum momento haverá uma entrada vazia, a menos que a chave já esteja presenta em `self.slots`. Ela computa o valor original de espalhamento e se o slot não estiver vazio, itera a função de `reshash` até apareça uma entrada vazia. Se um slot não vazio já contiver a chave, o valor antigo é substituído pelo novo. Deixamos como exercício a implementação da situação em que não há mais slots vazios.

Listing 3

```
def put(self,key,data):
  hashvalue = self.hashfunction(key,len(self.slots))

  if self.slots[hashvalue] == None:
    self.slots[hashvalue] = key
    self.data[hashvalue] = data
  else:
    if self.slots[hashvalue] == key:
      self.data[hashvalue] = data  #replace
    else:
      nextslot = self.rehash(hashvalue,len(self.slots))
      while self.slots[nextslot] != None and \
                      self.slots[nextslot] != key:
        nextslot = self.rehash(nextslot,len(self.slots))

      if self.slots[nextslot] == None:
        self.slots[nextslot]=key
        self.data[nextslot]=data
      else:
        self.data[nextslot] = data #replace
```

```
def hashfunction(self,key,size):
     return key%size

def rehash(self,oldhash,size):
    return (oldhash+1)%size
```

Da mesma forma, a função `get` (veja Listing 4) começa computando o valor inicial de espalhamento. Se o valor não estiver na entrada inicial, a função `rehash` é usada para encontrar a próxima posição possível. Note que a linha 15 garante que a busca irá terminar ao checar que não retornamos ao slot inicial. Se isso acontecer, visitamos todas as entradas possíveis e, portanto, o item não está presente.

Os métodos finais da classe `HashTable` proveem funcionalidade adicional ao dicionário. Fazemos uma sobrecarga dos métodos __getitem__ e __setitem__ para permitir o acesso usando `[]`. Isso significa que uma vez que a `HashTable` tenha sido criada, operador familiar de índice estará disponível. Deixamos os métodos restantes como exercícios.

Listing 4

```
1   def get(self,key):
2     startslot =
3  self.hashfunction(key,len(self.slots))
4
5     data = None
6     stop = False
7     found = False
8     position = startslot
9     while self.slots[position] != None and  \
10                          not found and not stop:
11        if self.slots[position] == key:
12          found = True
13          data = self.data[position]
14        else:
15
16  position=self.rehash(position,len(self.slots))
17          if position == startslot:
18              stop = True
19    return data
20
21  def __getitem__(self,key):
22      return self.get(key)
2
3   def __setitem__(self,key,data):
        self.put(key,data)
```

A sessão seguinte mostra a classe `HashTable` em ação. Primeiro, criamos uma tabela de dispersão e armazenamos alguns itens com chaves de inteiros e valores do tipo string.

```
>>> H=HashTable()
>>> H[54]="cat"
```

```
>>> H[26]="dog"
>>> H[93]="lion"
>>> H[17]="tiger"
>>> H[77]="bird"
>>> H[31]="cow"
>>> H[44]="goat"
>>> H[55]="pig"
>>> H[20]="chicken"
>>> H.slots
[77, 44, 55, 20, 26, 93, 17, None, None, 31, 54]
>>> H.data
['bird', 'goat', 'pig', 'chicken', 'dog', 'lion',
       'tiger', None, None, 'cow', 'cat']
```

A seguir, iremos acessar e modificar alguns itens na tabela de dispersão. Note que o valor para a chave 20 está sendo substituído.

```
>>> H[20]
'chicken'
>>> H[17]
'tiger'
>>> H[20]='duck'
>>> H[20]
'duck'
>>> H.data
['bird', 'goat', 'pig', 'duck', 'dog', 'lion',
       'tiger', None, None, 'cow', 'cat']
>> print(H[99])
None
```

Análise do Hashing

Dissemos anteriormente que no melhor caso, o hashing é uma técnica de busca de tempo constante, isto é,
O(1)
O(1). Contudo, devido às colisões, o número de comparações não é tipicamente tão simples. Embora uma análise completa do hashing esteja além do escopo deste texto, podemos mostrar alguns resultados bem conhecidos sobre o número aproximado de comparações necessárias para buscar um item.

A peça mais importante de informação de que precisamos para analisar o uso de uma tabela de dispersão é o fator de carga
λ
λ. Conceitualmente, se
λ
λ for pequeno, então há uma chance menor de colisões, o que significa que há mais chance de os itens estarem nas entradas a que pertencem. Se
λ

λ for grande, isto é, se a tabela estiver carregada, então haverá mais e mais colisões. Isso significa que a resolução de colisões será mais difícil, requerendo mais comparações para encontrar uma entrada vazia. Com o encademaneto, um número crescente de colisões significa um número crescente de itens em cada sequência (ou cadeia).
Assim como antes, teremos um resultado tanto para uma busca bem-sucedida quanto mal-sucedida. Para uma busca bem-sucedida, usando endereçamento aberto com sondagem linear, o número médio de comparações é de aproximadamente
½ (1+ 1/ 1−λ), enquanto a busca mal-sucedida resulta em no mesmo só que 1/1 - λ ao quadrado

Se usarmos o encadeamento, o número médio de comparações é
1+ λ /2 para o caso bem-sucedido e simplesmente λ comparações para busca mal-sucedida.