Tabela Hash: cont. pag 264 até 279.

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:

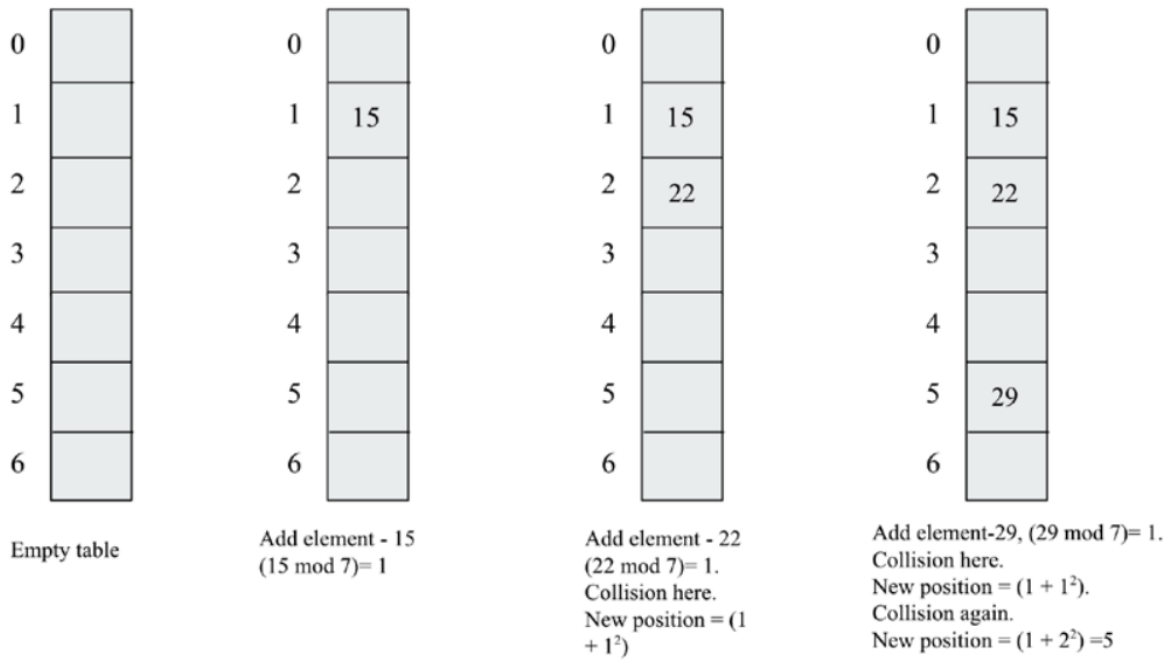

Figure 8.9: Example of collision resolution using quadratic probing

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.

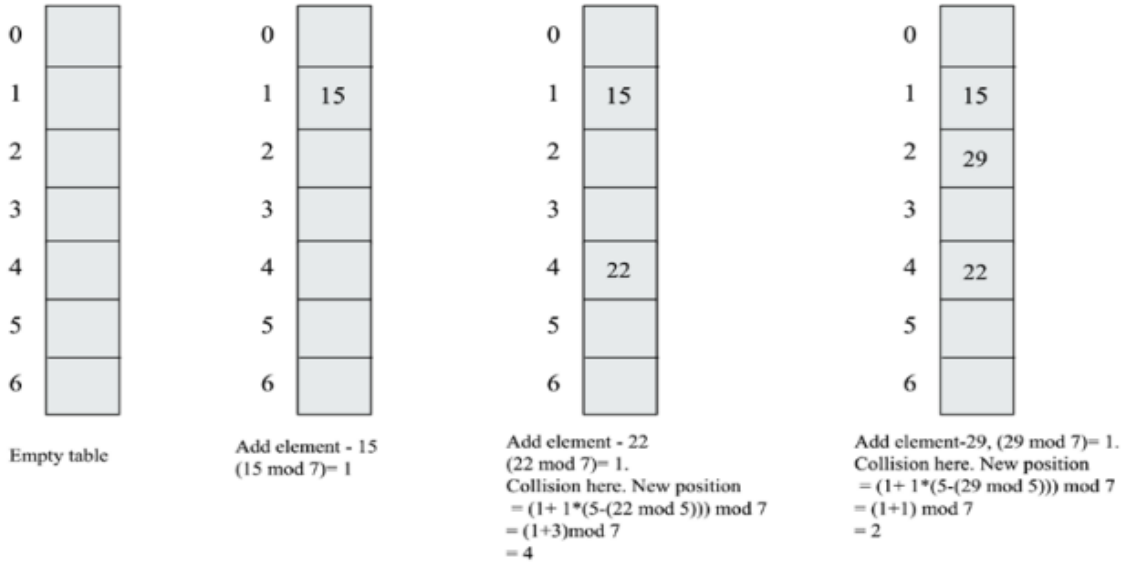

Figure 8.10: Example of collision resolution using double hashing

Vamos ver agora como podemos implementar a tabela de hash com a técnica de hashing duplo para resolver as colisões. Os métodos put_double_hashing() e get_double_hashing() são dados da seguinte forma, os quais devem ser definidos na classe HashTable.

O seguinte método h2() é usado para calcular a soma dos valores ordinais, já que em nossos exemplos, temos strings como elemento chave:

```
def h2(self, key):
    mult = 1
    hv = 0
    for ch in key:
        hv += mult * ord(ch)
        mult += 1
    return hv
```

Além disso, devemos redefinir a tabela de hash para incluir um número primo como uma variável que será usada no cálculo da função de hash secundária:

```
class HashTable:
    def __init__(self):
        self.size = 256
        self.slots = [None for i in range(self.size)]
        self.count = 0
        self.MAXLOADFACTOR = 0.65
        self.prime_num = 5
```

O seguinte código é projetado para inserir um elemento de dados e valor associado na tabela de hash e usar a técnica de hashing duplo no momento da colisão:

```
def put_double_hashing(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 * (self.prime_num - (self.h2(key) %
self.prime_num))) % self.size
            j = j+1
        if self.slots[h] == None:
            self.count += 1
        self.slots[h] = item
        self.check_growth()

    def get_double_hashing(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 * (self.prime_num - (self.h2(key) %
self.prime_num))) % self.size
            j = j + 1
        return None
```

O código acima dos métodos get_doubleHashing() e put_doubleHashing() são muito semelhantes à implementação dos métodos get() e put() que discutimos anteriormente, exceto pelas declarações que estão em negrito nos códigos

anteriores. As declarações em negrito mostram que, no momento da colisão, usamos a fórmula de técnica de hashing duplo para obter o próximo slot vazio na tabela de hash:

```
ht = HashTable()
ht.put_doubleHashing("good", "eggs")
ht.put_doubleHashing("better", "spam")
ht.put_doubleHashing("best", "cool")
ht.put_doubleHashing("ad", "donot")
ht.put_doubleHashing("ga", "collide")
ht.put_doubleHashing("awd", "hello")
ht.put_doubleHashing("addition", "ok")

for key in ("good", "better", "best", "worst", "ad", "ga"):
    v = ht.get_doubleHashing(key)
    print(v)
print("The number of elements is: {}".format(ht.count))
```

No código acima, primeiro inserimos sete elementos de dados diferentes junto com seus valores associados e, em seguida, pesquisamos e verificamos alguns itens de dados aleatórios na tabela de hash. A saída do código anterior é a seguinte:

```
eggs
spam
cool
none
donot
collide
The number of elements is: 7
```

Na saída acima, podemos observar que a string chave "worst" não está presente na tabela de hash, o que significa que a saída correspondente é None.

A sondagem linear leva a uma clusterização primária, enquanto a sondagem quadrática pode levar a uma clusterização secundária, enquanto a técnica de hashing duplo é um dos métodos mais eficazes para resolução de colisão, pois não produz nenhum cluster. A vantagem dessa técnica é que ela produz uma distribuição uniforme de registros na tabela de hash.

Nas técnicas de resolução de colisão de endereçamento aberto, procuramos outro slot vazio dentro da tabela de hash, como fizemos na sondagem linear, sondagem

quadrática e hashing duplo. "Fechado" em "hashing fechado" refere-se ao fato de que não saímos da tabela de hash, e cada registro é armazenado em uma posição de índice fornecida pela função hash, daí "hashing fechado" e "endereçamento aberto" são sinônimos.

Por outro lado, quando um registro é sempre armazenado em uma posição de índice fornecida pela função hash, isso é conhecido como a técnica de "endereçamento fechado" ou "hashing aberto". Aqui, "aberto" em "hashing aberto" refere-se ao fato de que estamos abertos para sair da tabela de hash por meio de uma lista separada onde os elementos de dados podem ser armazenados; por exemplo, o encadeamento separado é uma técnica de endereçamento fechado. Na próxima seção, discutiremos outra técnica de resolução de colisão - a técnica de encadeamento.

Encadeamento separado:

O encadeamento separado é outro método para lidar com o problema de colisão em tabelas hash. Ele resolve este problema permitindo que cada slot na tabela hash armazene uma referência a muitos itens na posição de uma colisão. Então, no índice de uma colisão, é permitido armazenar vários itens na tabela hash. No encadeamento, os slots na tabela hash são inicializados com listas vazias. Quando um elemento de dados é inserido, ele é adicionado à lista que corresponde ao valor hash desse elemento. Por exemplo, na Figura 8.11 a seguir, há uma colisão para as strings de chave "hello world" e "world hello". No caso do encadeamento, ambos os elementos de dados são armazenados usando uma lista na posição de índice dada pela função hash, ou seja, 92 no exemplo mostrado na Figura 8.11. Aqui está um exemplo para mostrar a resolução de colisão usando encadeamento:

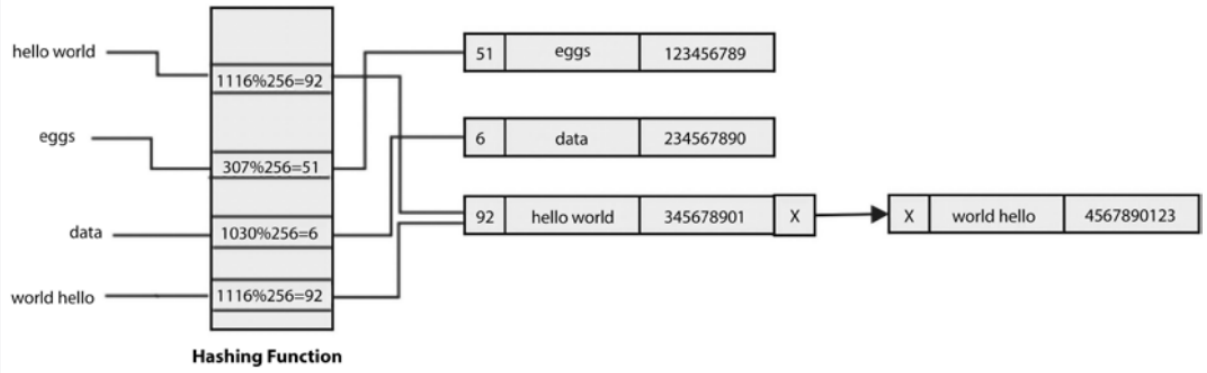

Figure 8.11: Example of collision resolution using chaining

Mais um exemplo é mostrado na Figura 8.12, em que se tivermos muitos elementos de dados que possuem um valor hash de 51, todos esses elementos seriam adicionados à lista que existe no mesmo slot da tabela hash:

Figure 8.12: More than one element having the same hash value stored in a list

O encadeamento evita conflitos permitindo que múltiplos elementos tenham o mesmo valor hash. Assim, não há limite em termos do número de elementos que podem ser armazenados em uma tabela hash, enquanto que no caso de técnicas de resolução de colisão de endereçamento aberto, tínhamos que fixar o tamanho da tabela, que precisamos aumentar posteriormente quando a tabela estiver preenchida. Além disso, a tabela hash pode armazenar 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 uma determinada localização de valor hash. Como um determinado slot tem muitos itens, a pesquisa pode se tornar muito lenta, pois temos que fazer uma busca linear através da lista até encontrarmos o elemento que tem a chave que queremos. Isso pode retardar a recuperação, o que não é bom, já que as tabelas hash devem ser eficientes. Assim, a complexidade de tempo de pior caso para a pesquisa em um algoritmo de encadeamento separado usando listas encadeadas é O(n), pois no pior caso, todos os itens serão adicionados a apenas uma posição de índice na tabela hash, e a busca por um item funcionará exatamente como uma lista encadeada. A Figura 8.13 a seguir demonstra uma pesquisa linear através de itens da lista até encontrarmos uma correspondência:

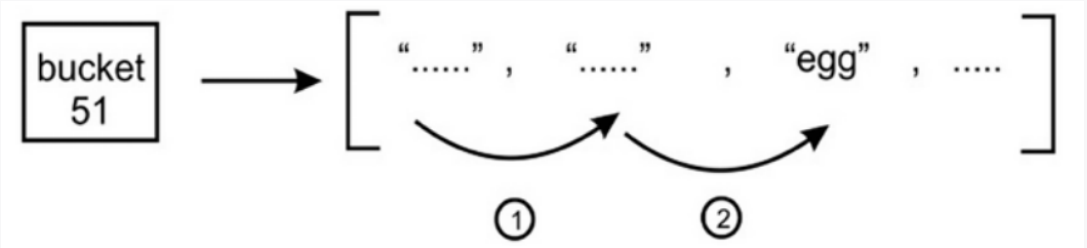

Figure 8.13: Demonstration of a linear search for the hash value of 51

Portanto, há um problema com a recuperação lenta de itens quando uma posição particular em uma tabela hash tem muitas entradas. Este problema pode ser resolvido usando outra estrutura de dados em vez de usar uma lista que possa realizar pesquisa e recuperação rapidamente. Existe uma boa escolha de usar árvores de busca binárias (BSTs), que fornecem uma recuperação rápida, como discutimos no capítulo anterior.

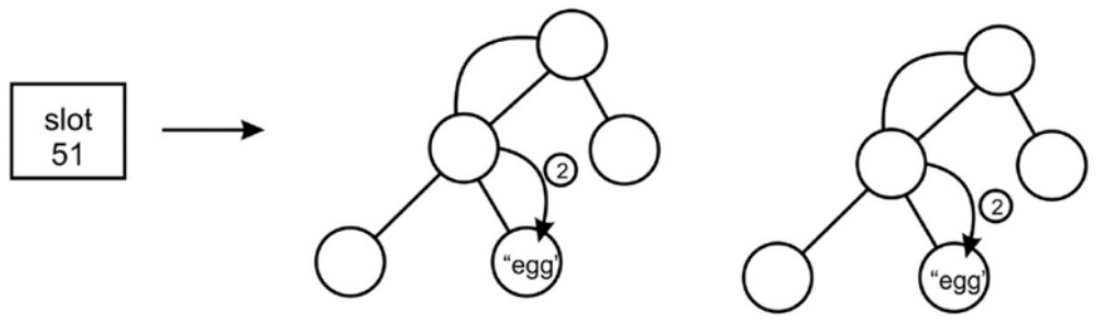

Figure 8.14: BST for a bucket for the hash value of 51

No diagrama anterior, o slot 51 contém uma BST, que usamos para armazenar e recuperar os itens de dados. No entanto, ainda teríamos um problema potencial - dependendo da ordem em que os itens foram adicionados à 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 nossa BST seja autoequilibrada.

Aqui está a implementação da tabela de hash com encadeamento separado. Em primeiro lugar, criamos uma classe Node para armazenar os pares chave-valor e um ponteiro para apontar para o próximo nó na lista encadeada:

```
class Node:
    def __init__(self, key=None, value=None):
        self.key = key
        self.value = value
        self.next = None
```

Em seguida, definimos a lista encadeada simples, cujos detalhes são fornecidos no Capítulo 4, Listas Ligadas. Aqui, definimos o método append() para adicionar um novo registro de dados à lista encadeada:

```
class SinglyLinkedList:

    def __init__ (self):
        self.tail = None
        self.head = None

    def append(self, key, value):
        node = Node(key, value)
        if self.tail:
            self.tail.next = node
            self.tail = node
```

```
        else:
            self.head = node
            self.tail = node
```

A seguir, definimos o método traverse(), que imprime todos os registros de dados com pares chave-valor. O método traverse() deve ser definido na classe SinglyLinkedList. Começamos pelo nó da cabeça e movemos os próximos nós enquanto iteramos pelo loop while:

```
def traverse(self):
        current = self.head
        while current:
              print("\"", current.key, "--", current.value, "\"")
            current = current.next
```

Em seguida, definimos um método de busca que corresponde à chave que desejamos procurar na lista encadeada. Se a chave corresponder a algum dos nós, o par chave-valor correspondente é impresso. O método search() deve ser definido na classe SinglyLinkedList:

```
def search(self, key):
        current = self.head
        while current:
            if current.key == key:
                    print("\"Record found:", current.key, "-", current.value, "\"")
                return True
            current = current.next
        return False
```

Depois de definirmos a lista encadeada e todos os métodos necessários, definimos a classe HashTableChaining, na qual inicializamos a tabela de hash com seu tamanho e todos os slots com uma lista encadeada vazia:

```
class HashTableChaining:
    def __init__(self):
        self.size = 6
        self.slots = [None for i in range(self.size)]
        for x in range(self.size) :
```

```
            self.slots[x] = SinglyLinkedList()
```

Em seguida, definimos a função de hash, ou seja, _hash(), semelhante ao que discutimos em seções anteriores:

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

Então, definimos o método put() para inserir um novo registro de dados na tabela de hash. Primeiro, criamos um nó com pares chave-valor e depois calculamos a posição do índice com base na função de hash. Em seguida, anexamos o nó ao final da lista encadeada associada à posição do índice fornecida. O método put() deve ser definido na classe HashTableChaining:

```
def put(self, key, value):
        node = Node(key, value)
        h = self._hash(key)
        self.slots[h].append(key, value)
```

A seguir, definimos o método get() para recuperar os elementos de dados dados o valor da chave da tabela de hash. Primeiro, calculamos a posição do índice usando a mesma função de hash que usamos ao adicionar os registros à tabela de hash e, em seguida, procuramos o registro de dados necessário na lista encadeada associada à posição do índice calculada. O método get() deve ser definido na classe HashTableChaining:

```
def get(self, key):
        h = self._hash(key)
        v = self.slots[h].search(key)
```

Por fim, podemos definir o método printHashTable(), que imprime a tabela de hash completa mostrando todos os registros da tabela de hash:

```
def printHashTable(self) :
        print("Hash table is :- \n")
        print("Index \t\tValues\n")
```

```
        for x in range(self.size) :
            print(x,end="\t\n")
            self.slots[x].traverse()
```

Podemos usar o seguinte código para inserir alguns registros de dados de exemplo na tabela hash e usamos a técnica de encadeamento para armazenar os dados. Em seguida, procuramos um registro de dados com a string de chave "best" e também imprimimos a tabela hash completa:

```
ht = HashTableChaining()
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.printHashTable()
```

A saída do código anterior é a seguinte:

```
Hash table is :-
Index             Values
0
1
2
" good - eggs "
3
" better - ham "
" ad - do not "
" ga - collide "
4
5
" best - spam "
" awd - do not "
```

A saída acima mostra como todos os registros de dados são armazenados em cada posição de índice na tabela hash. Podemos observar que vários registros de dados são armazenados na mesma posição de índice dada pela função hash.

Tabelas hash são estruturas de dados importantes para armazenar dados em pares de chave-valor, e podemos usar qualquer uma das técnicas de resolução de colisão, ou seja, endereçamento aberto ou encadeamento separado. As técnicas de endereçamento aberto são muito rápidas quando as chaves estão uniformemente distribuídas na tabela hash, mas há uma possível complicação na formação de clusters.

A técnica de encadeamento separado não tem o problema de clustering, mas pode ficar mais lenta quando todos os registros de dados são hashados para poucas posições de índice na tabela hash.

Tabelas de símbolos:

As tabelas de símbolos são usadas por compiladores e interpretadores para acompanhar os símbolos e diferentes entidades, como objetos, classes, variáveis e nomes de função, que foram declarados em um programa. As tabelas de símbolos geralmente são construídas usando tabelas hash, pois é importante recuperar eficientemente um símbolo da tabela.

Vamos olhar para um exemplo. Suponha que temos o seguinte código Python no arquivo symb.py:

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

Aqui, temos dois símbolos, "name" e "age". Cada símbolo tem um valor; por exemplo, o símbolo "name" tem o valor "Joe" e o símbolo "age" tem o valor "27". Uma tabela de símbolos permite que o compilador ou o interpretador procure esses valores. Portanto, os símbolos "name" e "age" se tornam chaves na tabela hash. Todas as outras informações associadas a eles se tornam o valor da entrada da tabela de símbolos.

Nos compiladores, as tabelas de símbolos podem ter outros símbolos, como nomes de funções e classes. Por exemplo, a função "greet()" e duas variáveis, ou seja, "name" e "age", são armazenadas na tabela de símbolos conforme mostrado na Figura 8.15:

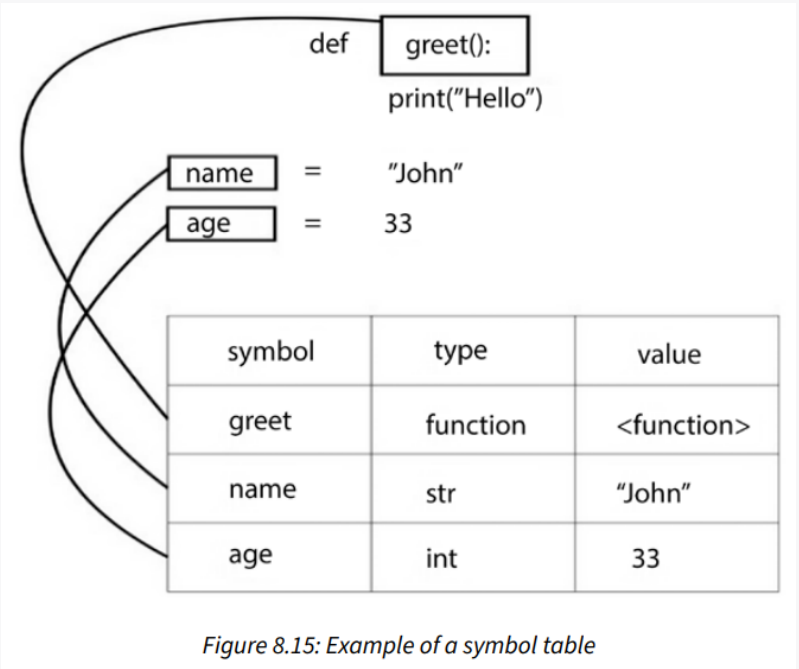

Figure 8.15: Example of a symbol table

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

Exercícios:

1. Há uma tabela hash com 40 slots e há 200 elementos armazenados na tabela. Qual será o fator de carga da tabela hash?
2. Qual é o tempo de busca no pior caso da técnica de encadeamento separado?
3. Suponha uma distribuição uniforme de chaves na tabela hash. Qual será a complexidade temporal para as operações de busca/inserção/exclusão?
4. Qual será a complexidade temporal no pior caso para remover caracteres duplicados de uma matriz de caracteres?