Algoritmos e Estruturas de Dados I

Hash Table - Python

Prof. Tiago Eugenio de Melo
tmelo@uea.edu.br

www.tiagodemelo.info

Observações

- O conteúdo dessa aula é parcialmente proveniente do Capítulo 11 do livro “*Data Structure and Algorithmic Thinking with Python*”.
- As palavras com a fonte `Courier` indicam uma palavra-reservada da linguagem de programação.

Introdução

Introdução

- O tipo dicionário de Python é implementado usando uma tabela hash.

Introdução

- O tipo dicionário de Python é implementado usando uma tabela hash.
- O dicionário em Python permite o uso de qualquer tipo de dados, **desde** que a chave possa ser mapeada para a tabela.

Introdução

- O tipo dicionário de Python é implementado usando uma tabela hash.
- O dicionário em Python permite o uso de qualquer tipo de dados, **desde** que a chave possa ser mapeada para a tabela.
- Essa restrição é uma limitação imposta pela linguagem de programação.

Hash Table TAD

Hash Table TAD

- Para se implementar uma tabela hash, nós devemos inicialmente decidir o tamanho da tabela.

Hash Table TAD

- Para se implementar uma tabela hash, nós devemos inicialmente decidir o tamanho da tabela.
- O HashMap TAD deve permitir armazenar qualquer quantidade de elementos (objetos).

Hash Table TAD

- Para se implementar uma tabela hash, nós devemos inicialmente decidir o tamanho da tabela.
- O HashMap TAD deve permitir armazenar qualquer quantidade de elementos (objetos).
  - E se a quantidade de objetos ultrapassar o tamanho inicial da tabela?

Hash Table TAD

- Para se implementar uma tabela hash, nós devemos inicialmente decidir o tamanho da tabela.
- O HashMap TAD deve permitir armazenar qualquer quantidade de elementos (objetos).
  - E se a quantidade de objetos ultrapassar o tamanho inicial da tabela?
    - Será necessário permitir que a tabela cresça (expanda), conforme a necessidade.

Hash Table TAD

- Para se implementar uma tabela hash, nós devemos inicialmente decidir o tamanho da tabela.
- O HashMap TAD deve permitir armazenar qualquer quantidade de elementos (objetos).
  - E se a quantidade de objetos ultrapassar o tamanho inicial da tabela?
    - Será necessário permitir que a tabela cresça (expanda), conforme a necessidade.
    - Assim, podemos começar com um tamanho relativamente pequeno (M = 7).

Hash Table TAD

- Para se implementar uma tabela hash, nós devemos inicialmente decidir o tamanho da tabela.
- O HashMap TAD deve permitir armazenar qualquer quantidade de elementos (objetos).
  - E se a quantidade de objetos ultrapassar o tamanho inicial da tabela?
    - Será necessário permitir que a tabela cresça (expanda), conforme a necessidade.
    - Assim, podemos começar com um tamanho relativamente pequeno (M = 7).
    - A tabela crescerá de tamanho (***rehashing***) a cada vez que o fator de carga estiver “alto”.

Hash Table TAD

Hash Table TAD

- Mas o que seria um fator de carga alto?

Hash Table TAD

- Mas o que seria um fator de carga alto?
  - Depende muito da estrutura que estamos adotando.

Hash Table TAD

- Mas o que seria um fator de carga alto?
  - Depende muito da estrutura que estamos adotando.
  - Um fator de carga entre ½ e 1/3 fornece uma boa performance no caso médio.

Hash Table TAD

- Mas o que seria um fator de carga alto?
  - Depende muito da estrutura que estamos adotando.
  - Um fator de carga entre ½ e 1/3 fornece uma boa performance no caso médio.
  - No nosso exemplo, vamos adotar um fator de carga de 2/3.

Hash Table TAD

Hash Table TAD

```
# Implementation of the Map ADT using closed hashing and a probe with double hashing.
from arrays import Array

    class HashMap :
    # Defines constants to represent the status of each table entry.

    UNUSED = None
    EMPTY = _MapEntry( None, None )

    # Creates an empty map instance.
    def __init__( self ):
        self._table = Array( 7 )
        self._count = 0
        self._maxCount = len(self._table) - len(self._table) // 3
```

Hash Table TAD

```
# Implementation of the Map ADT using closed hashing and a probe with double hashing.
from arrays import Array

    class HashMap :
    # Defines constants to represent the status of each table entry.

    UNUSED = None
    EMPTY = _MapEntry( None, None )

    # Creates an empty map instance.
    def __init__( self ):
        self._table = Array( 7 )
        self._count = 0
        self._maxCount = len(self._table) - len(self._table) // 3
```

Hash Table TAD

```
# Implementation of the Map ADT using closed hashing and a probe with double hashing.
from arrays import Array

    class HashMap :
    # Defines constants to represent the status of each table entry.

    UNUSED = None
    EMPTY = _MapEntry( None, None )

    # Creates an empty map instance.
    def __init__( self ):
        self._table = Array( 7 )
        self._count = 0
        self._maxCount = len(self._table) - len(self._table) // 3
```

array do hash table

Hash Table TAD

```
# Implementation of the Map ADT using closed hashing and a probe with double hashing.
from arrays import Array

    class HashMap :
    # Defines constants to represent the status of each table entry.

    UNUSED = None
    EMPTY = _MapEntry( None, None )

    # Creates an empty map instance.
    def __init__( self ):
        self._table = Array( 7 )
        self._count = 0
        self._maxCount = len(self._table) - len(self._table) // 3
```

array do hash table

Hash Table TAD

```
# Implementation of the Map ADT using closed hashing and a probe with double hashing.
from arrays import Array

    class HashMap :
    # Defines constants to represent the status of each table entry.

    UNUSED = None
    EMPTY = _MapEntry( None, None )

    # Creates an empty map instance.
    def __init__( self ):
        self._table = Array( 7 )
        self._count = 0
        self._maxCount = len(self._table) - len(self._table) // 3
```

array do hash table

#chaves armazenadas (atual)

Hash Table TAD

```
# Implementation of the Map ADT using closed hashing and a probe with double hashing.
from arrays import Array

    class HashMap :
    # Defines constants to represent the status of each table entry.

    UNUSED = None
    EMPTY = _MapEntry( None, None )

    # Creates an empty map instance.
    def __init__( self ):
        self._table = Array( 7 )
        self._count = 0
        self._maxCount = len(self._table) - len(self._table) // 3
```

array do hash table

#chaves armazenadas (atual)

Hash Table TAD

```
# Implementation of the Map ADT using closed hashing and a probe with double hashing.
from arrays import Array

    class HashMap :
    # Defines constants to represent the status of each table entry.

    UNUSED = None
    EMPTY = _MapEntry( None, None )

    # Creates an empty map instance.
    def __init__( self ):
        self._table = Array( 7 )
        self._count = 0
        self._maxCount = len(self._table) - len(self._table) // 3
```

array do hash table

#chaves armazenadas (atual)

fator de carga

Hash Table TAD

```
# Implementation of the Map ADT using closed hashing and a probe with double hashing.
from arrays import Array

    class HashMap :
    # Defines constants to represent the status of each table entry.

    UNUSED = None
    EMPTY = _MapEntry( None, None )

    # Creates an empty map instance.
    def __init__( self ):
        self._table = Array( 7 )
        self._count = 0
        self._maxCount = len(self._table) - len(self._table) // 3
```

array do hash table

#chaves armazenadas (atual)

fator de carga

Cada vez que a tabela é expandida, um novo valor de `maxCount` é calculado.

Hash Table TAD

```
# Implementation of the Map ADT using closed hashing and a probe with double hashing.
from arrays import Array

    class HashMap :
    # Defines constants to represent the status of each table entry.

    UNUSED = None
    EMPTY = _MapEntry( None, None )

    # Creates an empty map instance.
    def __init__( self ):
        self._table = Array( 7 )
        self._count = 0
        self._maxCount = len(self._table) - len(self._table) // 3
```

array do hash table

#chaves armazenadas (atual)

fator de carga

Cada vez que a tabela é expandida, um novo valor de `maxCount` é calculado.

Para o tamanho inicial de 7, o fator de carga será 5.

Hash Table TAD

Hash Table TAD

```
# Implementation of the Map ADT using closed hashing and a probe with double hashing.
from arrays import Array

    class HashMap :
    # Defines constants to represent the status of each table entry.

    UNUSED = None
    EMPTY = _MapEntry( None, None )

    # Creates an empty map instance.
    def __init__( self ):
        self._table = Array( 7 )
        self._count = 0
        self._maxCount = len(self._table) - len(self._table) // 3
```

Hash Table TAD

```
# Implementation of the Map ADT using closed hashing and a probe with double hashing.
from arrays import Array

    class HashMap :
    # Defines constants to represent the status of each table entry.

    UNUSED = None
    EMPTY = _MapEntry( None, None )

    # Creates an empty map instance.
    def __init__( self ):
        self._table = Array( 7 )
        self._count = 0
        self._maxCount = len(self._table) - len(self._table) // 3
```

Hash Table TAD

```
# Implementation of the Map ADT using closed hashing and a probe with double hashing.
from arrays import Array

    class HashMap :
    # Defines constants to represent the status of each table entry.

    UNUSED = None
    EMPTY = _MapEntry( None, None )

    # Creates an empty map instance.
    def __init__( self ):
        self._table = Array( 7 )
        self._count = 0
        self._maxCount = len(self._table) - len(self._table) // 3
```

Hash Table TAD

```
# Implementation of the Map ADT using closed hashing and a probe with double hashing.
from arrays import Array

    class HashMap :
    # Defines constants to represent the status of each table entry.

    UNUSED = None
    EMPTY = _MapEntry( None, None )

    # Creates an empty map instance.
    def __init__( self ):
        self._table = Array( 7 )
        self._count = 0
        self._maxCount = len(self._table) - len(self._table) // 3
```

Hash Table TAD

```
# Implementation of the Map ADT using closed hashing and a probe with double hashing.
from arrays import Array

    class HashMap :
    # Defines constants to represent the status of each table entry.

    UNUSED = None
    EMPTY = _MapEntry( None, None )

    # Creates an empty map instance.
    def __init__( self ):
        self._table = Array( 7 )
        self._count = 0
        self._maxCount = len(self._table) - len(self._table) // 3
```

tipos de entradas (flags)

Hash Table TAD

```
# Implementation of the Map ADT using closed hashing and a probe with double hashing.
from arrays import Array

class HashMap :
# Defines constants to represent the status of each table entry.

    UNUSED = None
    EMPTY = _MapEntry( None, None )

    # Creates an empty map instance.
    def __init__( self ):
        self._table = Array( 7 )
        self._count = 0
        self._maxCount = len(self._table) - len(self._table) // 3
```

indica que a chave (key) ainda não foi usada para armazenar objetos.

dummy

tipos de entradas (flags)

Hash Table TAD

```
# Implementation of the Map ADT using closed hashing and a probe with double hashing.
from arrays import Array

    class HashMap :
    # Defines constants to represent the status of each table entry.

    UNUSED = None
    EMPTY = _MapEntry( None, None )

    # Creates an empty map instance.
    def __init__( self ):
        self._table = Array( 7 )
        self._count = 0
        self._maxCount = len(self._table) - len(self._table) // 3
```

indica que a chave (key) ainda não foi usada para armazenar objetos.

espaço já havia sido usado, mas ele foi apagado.

dummy

tipos de entradas (flags)

Hash Table TAD

```
# Implementation of the Map ADT using closed hashing and a probe with double hashing.
from arrays import Array

    class HashMap :
    # Defines constants to represent the status of each table entry.

    UNUSED = None
    EMPTY = _MapEntry( None, None )

    # Creates an empty map instance.
    def __init__( self ):
        self._table = Array( 7 )
        self._count = 0
        self._maxCount = len(self._table) - len(self._table) // 3
```

indica que a chave (key) ainda não foi usada para armazenar objetos.

espaço já havia sido usado, mas ele foi apagado.

dummy

tipos de entradas (flags)

```
# Storage class for holding the key/value pairs.
class _MapEntry :
  def __init__( self, key, value ):
    self.key = key
    self.value = value
```

Hash Table TAD

Hash Table TAD

- Vamos adotar duas funções de *hash*:

Hash Table TAD

- Vamos adotar duas funções de *hash*:
  - Uma principal para mapear o elemento para sua posição;

Hash Table TAD

- Vamos adotar duas funções de *hash*:
  - Uma principal para mapear o elemento para sua posição;
  - Uma secundária para realizar o duplo *hashing*;

Hash Table TAD

- Vamos adotar duas funções de *hash*:
  - Uma principal para mapear o elemento para sua posição;
  - Uma secundária para realizar o duplo *hashing*;
- Função principal:

Hash Table TAD

- Vamos adotar duas funções de *hash*:
  - Uma principal para mapear o elemento para sua posição;
  - Uma secundária para realizar o duplo *hashing*;
- Função principal:
  - Método da divisão:

Hash Table TAD

- Vamos adotar duas funções de *hash*:
  - Uma principal para mapear o elemento para sua posição;
  - Uma secundária para realizar o duplo *hashing*;
- Função principal:
  - Método da divisão:
  - Usaremos a função nativa de Python: `hash ( )`

Hash Table TAD

- Vamos adotar duas funções de *hash*:
  - Uma principal para mapear o elemento para sua posição;
  - Uma secundária para realizar o duplo *hashing*;
- Função principal:
  - Método da divisão:
  - Usaremos a função nativa de Python: `hash ( )`
    - Essa função retorna um valor inteiro para uma dada chave (key) de entrada.

Hash Table TAD

- Vamos adotar duas funções de *hash*:
  - Uma principal para mapear o elemento para sua posição;
  - Uma secundária para realizar o duplo *hashing*;
- Função principal:
  - Método da divisão:
  - Usaremos a função nativa de Python: `hash ( )`
    - Essa função retorna um valor inteiro para uma dada chave (key) de entrada.
    - Esse valor pode ser usado no método da divisão.

Hash Table TAD

- Vamos adotar duas funções de *hash*:
  - Uma principal para mapear o elemento para sua posição;
  - Uma secundária para realizar o duplo *hashing*;
- Função principal:
  - Método da divisão:
  - Usaremos a função nativa de Python: `hash ( )`
    - Essa função retorna um valor inteiro para uma dada chave (key) de entrada.
    - Esse valor pode ser usado no método da divisão.
    - Esse valor pode ser negativo ou ir além do valor do tamanho da tabela (M).

Hash Table TAD

- Vamos adotar duas funções de *hash*:
  - Uma principal para mapear o elemento para sua posição;
  - Uma secundária para realizar o duplo *hashing*;
- Função principal:
  - Método da divisão:
  - Usaremos a função nativa de Python: `hash ( )`
    - Essa função retorna um valor inteiro para uma dada chave (key) de entrada.
    - Esse valor pode ser usado no método da divisão.
    - Esse valor pode ser negativo ou ir além do valor do tamanho da tabela (M).

```
h(key) = |hash(key)| % M
```

Hash Table TAD

Hash Table TAD

- Função Secundária:

Hash Table TAD

- Função Secundária:
  - Método da divisão:

Hash Table TAD

- Função Secundária:
  - Método da divisão:
  - Usaremos a função nativa de Python: `hash ( )`

Hash Table TAD

- Função Secundária:
  - Método da divisão:
  - Usaremos a função nativa de Python: `hash ( )`

```
h2(key) = 1 + |hash(key)| % (M - 2)
```

Hash Table TAD

- Funções de *hashing*:

```
# The main hash function for mapping keys to table entries.
def _hash1( self, key ):
    return abs( hash(key) ) % len(self._table)

# The second hash function used with double hashing probes.
def _hash2( self, key ):
    return 1 + abs( hash(key) ) % (len(self._table) - 2)
```

Hash Table TAD

Hash Table TAD

- Função de busca

Hash Table TAD

- Função de busca

```
# Finds the slot containing the key or where the key can be added.
# forInsert indicates if the search is for an insertion, which locates
# the slot into which the new key can be added.

def _findSlot( self, key, forInsert ):
    # Compute the home slot and the step size.
    slot = self._hash1( key )
    step = self._hash2( key )

    # Probe for the key.
    M = len(self._table)
    while self._table[slot] is not UNUSED:
        if forInsert and (self._table[slot] is UNUSED or self._table[slot] is EMPTY):
            return slot
        elif not forInsert and (self._table[slot] is not EMPTY and self._table[slot].key == key) :
            return slot
        else :
            slot = (slot + step) % M
```

Hash Table TAD

- Função de busca

```
# Finds the slot containing the key or where the key can be added.
# forInsert indicates if the search is for an insertion, which locates
# the slot into which the new key can be added.

def _findSlot( self, key, forInsert ):
    # Compute the home slot and the step size.
    slot = self._hash1( key )
    step = self._hash2( key )

    # Probe for the key.
    M = len(self._table)
    while self._table[slot] is not UNUSED:
        if forInsert and (self._table[slot] is UNUSED or self._table[slot] is EMPTY):
            return slot
        elif not forInsert and (self._table[slot] is not EMPTY and self._table[slot].key == key) :
            return slot
        else :
            slot = (slot + step) % M
```

`forInsert`

- `True`: a busca é para localizar onde uma nova chave possa ser inserida.
- `False`: uma busca é executada… ou o índice da entrada é retornado ou None é retornado, indicando que a chave não está na tabela.

Hash Table TAD

- Inserção (add)

```
# Adds a new entry to the map if the key does not exist. Otherwise, the
# new value replaces the current value associated with the key.
def add( self, key, value ):
    if key in self :
        slot = self._findSlot( key, False )
        self._table[slot].value = value
        return False
    else :
        slot = self._findSlot( key, True )
        self._table[slot] = _MapEntry( key, value )
        self._count += 1
        if self._count == self._maxCount :
            self._rehash()
        return True
```

Hash Table TAD

Hash Table TAD

- Inserção (add)

Hash Table TAD

- Inserção (add)
  - O método é usado duas vezes:

Hash Table TAD

- Inserção (add)
  - O método é usado duas vezes:
    - Determinar se a chave está na tabela.

Hash Table TAD

- Inserção (add)
  - O método é usado duas vezes:
    - Determinar se a chave está na tabela.
      - True

Hash Table TAD

- Inserção (add)
  - O método é usado duas vezes:
    - Determinar se a chave está na tabela.
      - True
        - Localizar a chave.

Hash Table TAD

- Inserção (add)
  - O método é usado duas vezes:
    - Determinar se a chave está na tabela.
      - True
        - Localizar a chave.
        - Modificar o seu valor.

Hash Table TAD

- Inserção (add)
  - O método é usado duas vezes:
    - Determinar se a chave está na tabela.
      - True
        - Localizar a chave.
        - Modificar o seu valor.
      - False

Hash Table TAD

- Inserção (add)
  - O método é usado duas vezes:
    - Determinar se a chave está na tabela.
      - True
        - Localizar a chave.
        - Modificar o seu valor.
      - False
        - A chave não está na tabela.

Hash Table TAD

- Inserção (add)
  - O método é usado duas vezes:
    - Determinar se a chave está na tabela.
      - True
        - Localizar a chave.
        - Modificar o seu valor.
      - False
        - A chave não está na tabela.
        - Chamar o método __findSlot__ para localizar o próximo slot disponível.

Hash Table TAD

- Inserção (add)
  - O método é usado duas vezes:
    - Determinar se a chave está na tabela.
      - True
        - Localizar a chave.
        - Modificar o seu valor.
      - False
        - A chave não está na tabela.
        - Chamar o método __findSlot__ para localizar o próximo slot disponível.
        - Finalmente, nós checamos a quantidade e determinamos se ela excede o fator de carga.

Hash Table TAD

- Rehashing

```
# Rebuilds the hash table.
def _rehash( self ) :
    # Create a new larger table.
    origTable = self._table
    newSize = len(self._table) * 2 + 1
    self._table = Array( newSize )

    # Modify the size attributes.
    self._count = 0
    self._maxCount = newSize - newSize // 3

    # Add the keys from the original array to the new table.
    for entry in origTable :
        if entry is not UNUSED and entry is not EMPTY :
            slot = self._findSlot( key, True )
            self._table[slot] = entry
            self._count += 1
```

Hash Table TAD

Hash Table TAD

- Rehashing

Hash Table TAD

- Rehashing
  - Primeiro passo é criar um array maior.

Hash Table TAD

- Rehashing
  - Primeiro passo é criar um array maior.
  - Por simplicidade: `M * 2 + 1`

Hash Table TAD

- Rehashing
  - Primeiro passo é criar um array maior.
  - Por simplicidade: `M * 2 + 1`
    - Podemos pensar em soluções melhores, tais como uso de números primos!

Hash Table TAD

- Rehashing
  - Primeiro passo é criar um array maior.
  - Por simplicidade: `M * 2 + 1`
    - Podemos pensar em soluções melhores, tais como uso de números primos!
  - O array inicial é armazenado em uma variável temporária: `origTable`

Hash Table TAD

- Rehashing
  - Primeiro passo é criar um array maior.
  - Por simplicidade: `M * 2 + 1`
    - Podemos pensar em soluções melhores, tais como uso de números primos!
  - O array inicial é armazenado em uma variável temporária: `origTable`
  - As variáveis `count` e `maxCount` também precisam ser resetadas.

Hash Table TAD

- Rehashing
  - Primeiro passo é criar um array maior.
  - Por simplicidade: `M * 2 + 1`
    - Podemos pensar em soluções melhores, tais como uso de números primos!
  - O array inicial é armazenado em uma variável temporária: `origTable`
  - As variáveis `count` e `maxCount` também precisam ser resetadas.
  - Finalmente, os pares <chave,valor> são adicionados ao novo array (um por vez).

Exercícios

- Existe ordenação dos elementos de um hash de Python? Explique a sua resposta.