Tema 9: Estructuras asociativas

Héctor Xavier Limón Riaño

May 31, 2024

Contents

1 Introducción **1**

2 Diccionarios Python **2**
2.1 Operaciones 3
2.1.1 Crear diccionario vacío 3
2.1.2 Longitud de diccionario 3
2.1.3 Agregar nueva llave-valor 3
2.1.4 Cambiar el valor asociado a una llave existente 3
2.1.5 Recuperar valor a partir de llave 4
2.1.6 Regresar llaves 4
2.1.7 Recorrer diccionario 5
2.1.8 Determinar si una llave pertenece al diccionario 5
2.1.9 Borrar un par llave-valor 6

3 Funcionamiento interno **6**
3.1 Funciones de hash 6
3.1.1 Función de hash en Python 8

1 Introducción

- A las estructuras asociativas se les llama de diversas formas: mapas de hash, diccionarios, tablas de hash
- Muchos lenguajes de programacion incluyen esta estructura en

su API

- Es una estructura muy eficiente, el acceso a los elementos es en la mayoria de los casos aleatorio

- Es una de las estructuras mas poderosas y utiles
- Son estructuras indexadas, pero a diferencia de una lista, el índice pude ser de cualquier tipo no mutable
- Los elementos de una estructura asociativa se componen de dos partes:
  - Llave: es el índice con el que se puede recuperar el valor
  - Valor: es el valor en si que se desea almacenar
- La idea es que a partir de una llave dada pueda almacenarse o

recuperarse un valor de forma aleatoria (igual que en un arreglo)

2 Diccionarios Python

- En `Python` la estructura asociativa estándar son los `diccionarios`

```
diccionario = {'llave1': 'valor 1', 'llave2': 'valor 2'} # definición literal
print(diccionario['llave1']) # recuperar

diccionario['llave3'] = 'valor3' # establecer
diccionario['llave1'] = 22 # mutable
print(diccionario['llave3'])
```

```
valor 1
valor3
```

- Son estructuras mutables y no lineales
- La llave sólo puede ser de un tipo no mutable como cadenas, tuplas, enteros, etc. (se explicará porqué más adelante)
- El valor puede ser de cualquier tipo

```
d1 = {3: 'tres'} # OK
d2 = {(1,): 'uno'} # OK
d3 = {[1, 2]: 'dos'} # error la llave es list
```

- Considerar que en estas estructuras los elementos no se ordenan de manera particular (como si pasa en las listas), por lo que no se puede predecir su orden interno o intentar ordenar (mediante `sorted` por ejemplo)
- Si imprimes un diccionario, es posible que los elementos no aparezcan en el orden en que los insertaste

2.1 Operaciones

2.1.1 Crear diccionario vacío

- Dos formas

```
d1 = {} # forma literal
d2 = dict() # usando función de conversión

print(type(d1))
print(type(d2))
```

```
<class 'dict'>
<class 'dict'>
```

2.1.2 Longitud de diccionario

- Se refiere a cuántos pares llave-valor tiene
- Como en otras estructuras se utiliza la función `len`

```
d = {'a': 1, 'b': 2, 'c': 3}
print(len(d))
```

```
3
```

2.1.3 Agregar nueva llave-valor

```
d1 = {'a': 1, 'b': 2}
d1['c'] = 3
print(d1)
```

```
{'a': 1, 'b': 2, 'c': 3}
```

2.1.4 Cambiar el valor asociado a una llave existente

- Es exactamente igual que agregar una nueva llave-valor

```
d1 = {'a': 1, 'b': 2}
d1['a'] = 66
print(d1)
```

```
{'a': 66, 'b': 2}
```

2.1.5 Recuperar valor a partir de llave

- Hay dos formas comunes:
  - Directa usando `[]`
  - Indirecta a partir de función `get`
- La forma directa tiene la desventaja de que si la llave no existe se lanza una excepción
- Mediante el método `get` se puede pasar un parámetro opcional que establece el valor por defecto a regresar si la llave no existe
- En general siempre es mejor recuperar con método `get`

```
d = {'a': 1, 'b': 2, 'c': 3}

print(d['a']) # forma directa
print(d.get('a')) # indirecta
print(d.get('d', -1)) # valor por defecto
print(d.get('b', -1)) # si existe no pasa nada

1
1
-1
2
```

2.1.6 Regresar llaves

- Regresa todas las llaves en una estructura similar a una lista ($\text{dict}_{\text{keys}}$)
- De ser necesario se puede convertir el resultado a una lista normal mediante `list`
- Se usa el método `keys`
- Útil para recorrer el diccionario (visto más adelante)

```
d = {'a': 1, 'b': 2, 'c': 3}
print(d.keys())

print(list(d.keys()))

dict_keys(['a', 'b', 'c'])
['a', 'b', 'c']
```

2.1.7 Recorrer diccionario

- Coma tal no se puede hacer directamente dado que la estructura no es lineal
- Una forma es obteniendo primero la lista de llaves

```
d = {'a': 1, 'b': 2, 'c': 3}

for llave in d.keys():
    print(d[llave]) # no hay riesgo de que no exista
```

```
1
2
3
```

- Se puede también recorrer tanto llaves y valores a la vez mediante el método `items`
- Dicho método regresa una lista de tuplas con los pares llave-valor

```
d = {'a': 1, 'b': 2, 'c': 3}

print(d.items())

for k, v in d.items():
    print('llave %s, valor %s' % (k, v))
```

```
dict_items([('a', 1), ('b', 2), ('c', 3)])
llave a, valor 1
llave b, valor 2
llave c, valor 3
```

2.1.8 Determinar si una llave pertenece al diccionario

- Se puede usar el operador `in`

```
d = {'a': 1, 'b': 2, 'c': 3}
llave = 'b'
print(llave in d.keys())
```

```
True
```

2.1.9 Borrar un par llave-valor

- Se puede como en otros casos con la función `del`
- En general no es tan común o necesario borrar cosas

```
d = {'a': 1, 'b': 2, 'c': 3}
del(d['a'])

print(d)
```

```
{'b': 2, 'c': 3}
```

3 Funcionamiento interno

- Un hash-map se implementa tradicionalmente a partir de un arreglo interno (que puede ser de dos dimensiones, a lo que se le llama una tabla)
- Es en el arreglo interno donde en realidad se guardan los elementos
- El arreglo interno permite que el accesos a los elementos sea aleatorio
- La idea es convertir la llave de un elemento a un ındice numérico del arreglo interno
- Esta conversion se logra mediante una funcion de `hash`
- Con el índice numérico se recupera directamente el valor

3.1 Funciones de hash

- A este tipo de funciones tambien se les llama de resumen
- Es una funcion que recibe algun tipo de objeto (para nuestro caso el tipo de la llave) y regresa un valor en un rango predefinido (normalmente un numero)
- Por ejemplo. Imaginemos una funcion de hash que recibe una cadena y lo que hace es devolver un valor entre 0 y 99.
  - La misma cadena siempre genera el mismo valor
  - Cualquier cadena que le pasemos sin importar su extension generara un numero entre 0 y 99

  - Como el dominio de la funcion es un conjunto infinito pero el codominio es un conjunto finito es obvio que varios (de hecho un numero infinito) de los valores del dominio se mapearan con el mismo valor del codominio. Al hecho anterior se le llama `Colision`
  - Entre mas grande sea la aridad del codominio menor sera la probabilidad de que dos elementos colisionen entre si

- En una estructura asociativa deben definirse:
  - Una funcion hash para convertir de llave a índice
  - Una polıtica de manejo de colisiones
    * Una política común es encadenamiento, esto es, se usa una lista ligada de los valores que colisionaron en la misma posición (de esta forma el arreglo interno guarda realmente listas ligadas)

- Existen diversas funciones de hash con diversos propositos, en este curso nos concentraremos en funciones de hash que sean capaces de generar un numero entre 0 y el numero total -1 de elementos de nuestro arreglo interno

- La funcion deberıa de dispersar de la forma mas equitativa posible los valores generados, de tal manera que se minimicen las colisiones

- Ejemplos de funciones de hash
  - Módulo
    * Se recomienda que el módulo sea un número primo (dispersión más equitativa)

```
def hashear(llave: str, modulo: int) -> int:
    """
    Regresa el hash de una llave de tipo cadena
    para el valor de módulo dado

    llave: str
    modulo: int
    returns: int, hash calculado
    """
    suma = 0
    for c in llave:
```

```
        suma += ord(c)
    return suma % modulo

modulo = 1013 # número primo
print(hashear('hola mundo', modulo))
```

```
999
```

- Función de Horner
  - Es una mejor función que el módulo, aplicable normalmente a cadenas
  - Utiliza descomposición de polinomios
  - Mejora la distribución de los valores de hash en comparación a módulo
  - No se verá su implementación

3.1.1 Función de hash en Python

- Python utiliza la función `hash` para determinar el valor numérico de las llaves
- Con este valor resuelve el índice correspondiente en el diccionario (para encontrar la casilla correspondiente, también llamada `bucket`)

```
print(hash(33))
print(hash('hola mundo'))
print(hash((1, 2)))
```

```
33
-1365708791437619780
-3550055125485641917
```

- Dependiendo del tipo de la llave, `Python` utiliza distintos algoritmos de hash
- Sólo los tipos inmutables son hasheables:

```
hash([1, 2, 3]) # error, unhashable
```

- Los tipos mutables no son hasheables debido a que su valor interno puede cambiar, lo cual afecta el cálculo de hash
- Dado que el cálculo de hash se hace cada vez que se usan las llaves en un diccionario, se necesita que sea un cálculo determinista (siempre de lo mismo), de lo contrario no se puede recuperar el valor (se encontraría otro bucket)
- Y por esta razón no se pueden usar tipos mutables como llaves