파이썬으로 배우는 데이터 구조

한동대학교
전산전자공학부
김영섭 교수

학습 목표

해시맵(Hashmap)의 ADT를

이해하고 구현할 수 있다

Data Structures in Python
Chapter 6

- Hash Table
- Collision Resolution
- Double Hashing & Rehashing
- **HashMap Coding**

Agenda & Readings

- Hash map Implementation
  - Map Abstract Data Type(ADT)
  - Map ADT Implementation
  - Using the [ ] syntax
  - Using the del Operator

- Reference:
  - Problem Solving with Algorithms and Data Structures
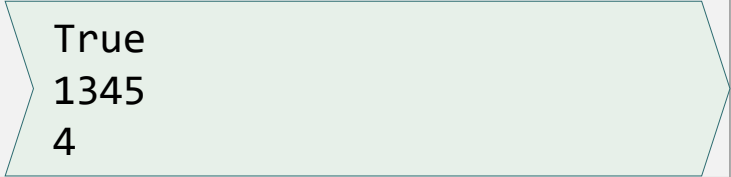 - Chapter 5 - Hashing

Map Abstract Data Type

- Map data structure:
    - It is a Non-Linear Data Structure which stores data in key-value pairs.
    - The keys are unique, and values can have duplicate entries.
    - It doesn't have any order or sequence like Linked List. Data is stored randomly.

- Examples:
    - Python - Dictionary
    - Java - HashTable, HashMap, LinkedHashMap
    - C++ - map, unordered_map

Map Abstract Data Type

- Operations of a Map ADT:
  - put(key, value)
  - get(key)
  - del map[key]
  - len()
  - in   # contains a given key
- The Python dictionary stores key–value pairs where the key is unique.
  The key is used to look up the associated data value.
  The Python dictionary is an implementation of the Map ADT.
- Example:

```
phone_ext = {'Lee':2410,'Sam':1131,"Pete":2830, "John":1345}
phone_ext["Liz"] = 1123
print('Sam' in phone_ext)
print(phone_ext["John"])
del phone_ext["Sam"]
print(len(phone_ext))
```

```
True
1345
4
```

Map ADT – An Implementation

- We will use two parallel Python lists, one for the slot numbers corresponding to the keys and one for the associated data. We are using linear probing to resolve collisions.

  - Example: The table size is 11 initially; **hash(key) = key % 11**

	0	1	2	3	4	5	6	7	8	9	10
slot	None	None	None	None	None	None	None	None	None	None	None

	0	1	2	3	4	5	6	7	8	9	10
data	None	None	None	None	None	None	None	None	None	None	None

- After all the items have been inserted:

	0	1	2	3	4	5	6	7	8	9	10
slot	77	44	55	20	26	93	17	None	None	31	54

	0	1	2	3	4	5	6	7	8	9	10
data	'bird'	'goat'	'pig'	'bear'	'dog'	'lion'	'ox'	None	None	'cow'	'cat'

```
ht = HashTable()
ht[54] = "cat"
ht[26] = "dog"
ht[93] = "lion"
ht[17] = "ox"
ht[77] = "bird"
ht[31] = "cow"
ht[44] = "goat"
ht[55] = "pig"
ht[20] = "bear"
```

Map ADT - An Implementation

- In this implementation we are using the hash function:

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

  - Whenever we add an item, we need to call the hash function:

```
hashcode = self.hashfunction(key)
```

- We will resolve collisions using **linear probing**, i.e., a step size of 1.

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

  - Whenever there is a collision, we need to get the next slot to try:

```
nextslot = self.rehash(nextslot)
```

Map ADT - An Implementation

- Create the two Python lists and set the size of the mapping:

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

    #define the get() and put() methods

        ...

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

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

```
put(key, value)
get(key)
del map[key]
len()
in        #contains
```

Map ADT - An Implementation

- Getting the associated value of an entry in the hash table:

```python
def get(self, key):
    startslot = self.hashfunction(key)
    position = startslot

    while self.slot[position] != None:
        if self.slot[position] == key:       # key found
            return self.data[position]       # return associated data
        else:
            position = self.rehash(position)
            if position == startslot:        # all slots in hash table searched
                return None                  # key not in table
    return None                              # empty slot – key not in table
```

```
put(key, value)
get(key)
del map[key]
len()
in        #contains
```

Map ADT - An Implementation

- Putting an entry (key-value pair) into the hash table:

```python
def put(self, key, data):
    hashcode = self.hashfunction(key)
    if self.slot[hashcode] == None:
        self.slot[hashcode] = key              # Put the key and associated data into
        self.data[hashcode] = data             # the lists
    elif self.slot[hashcode] == key:
        self.data[hashcode] = data             # Replace the associated data
    else:
        nextslot = self.rehash(hashcode)
        while self.slot[nextslot] != None and self.slot[nextslot] != key:
            nextslot = self.rehash(nextslot)
            if nextslot == hashcode:           # Hash table full, cannot add data
                return None
        if self.slot[nextslot] == None:        # Put the key and associated data into
            self.slot[nextslot] = key          # the lists
            self.data[nextslot] = data
        else:
            self.data[nextslot] = data         # Replace the associated data
```

Map ADT - An Implementation

- Similar to the Python dictionary data type, we want to allow applications to use the special [ ] syntax, i.e.:
  - To assign a new mapping: **ht[54] = "cat"**
  - To access the associated value in a mapping: **value = ht[54]**

```
def __setitem__(self, key, data):
    self.put(key, data)          #refers to the put() method


def __getitem__(self, key):
    return self.get(key)          #refers to the get() method
```

Map ADT - An Implementation

- The implementation now allows the use of the [ ] syntax.

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

    def put(self, key, data):
    ...
    def get(self, key):
    ...

    def __setitem__(self, key, data):
        self.put(key, data)        #refers to the put() method

    def __getitem__(self, key):
        self.get(key)              #refers to the get() method
```

```python
ht = HashTable()
ht[54] = "cat"
print(ht[54])
```

HashTable – Deleting a key-value pair

- Deleting a value is non–trivial because of collisions (see next slides).
- **Case 1: key is NOT in the table:**
    - Apply hash function. The slot is either 'None'(we can return) or occupied by another key. If occupied, we look sequentially(linear probing) until we find an element which is 'None'.
    - **Example:** Find hash[23] – we apply the hash function h(k)%11, and look in slot 1, then in slots 2, 3, 4, 5, 6, 7. Since slot 7 is 'None' we know the key 23 is not in the table and we do not need to look any further.

	0	1	2	3	4	5	6	7	8	9	10
slot	77	44	55	20	26	93	17	None	None	31	54

	0	1	2	3	4	5	6	7	8	9	10
data	'bird'	'goat'	'pig'	'bear'	'dog'	'lion'	'ox'	None	None	'cow'	'cat'

HashTable – Deleting a key-value pair

- Deleting a value is non-trivial because of collisions (see next slides).
- **Case 2: key is in the table:**
    - Assume we wish to delete **hash[55].** We apply the hash function and look in slot 0 (since **55%11 = 0**), then we look in slots 1, 2. We find key 55 at 2 and delete it.

	0	1	2	3	4	5	6	7	8	9	10
slot	77	44	55 None	20	26	93	17	None	None	31	54

	0	1	2	3	4	5	6	7	8	9	10
data	'bird'	'goat'	None	'bear'	'dog'	'lion'	'ox'	None	None	'cow'	'cat'

- **Now, what happens if we now wish to find key 20?** (20%11=9)
  Because of collisions it **has been inserted into slot 3**.
  But because slot 2 is now empty **(after deleting 55),** we **cannot** find key 20 anymore.

HashTable – Deleting a key-value pair

- We will need to use a dummy value for elements which have been deleted. In the constructor we can set **self.deleted** to be the **'\0'** null character.

```python
class HashTable:
    def __init__(self):
        self.size = 11
        self.slot = [None] * self.size
        self.data = [None] * self.size
        self.deleted = '\0'
```

HashTable - Deleting a key-value pair

- Let us code the **delete()** method to use "self.deleted" to indicate the deletion:

```python
def delete(self, key):
    startslot = self.hashfunction(key)
    position = startslot
    key_in_slot = self.slot[position]

    while key_in_slot != None:
        if key_in_slot == key:
            self.slot[position] = self.deleted
            self.data[position] = self.deleted
            return None
        else:
            position = self.rehash(position)
            key_in_slot = self.slot[position]
            if position == startslot:
                return None
```

> continue to search even if the slot contains **self.deleted.**
> Only stops if slot is None.

> Key not in table – do nothing and return

HashTable – Deleting a key-value pair

- The __delitem__( ) allows the use of the del operator.

```python
def delete(self, key):
    ... # see previous slide

def __delitem__(self, key):
    return self.delete(key)
```

```python
ht = HashTable()
ht[54] = "cat"
ht[31] = "cow"
ht[44] = "goat"
del ht[44]
del ht[54]
```

HashTable – Updating put() function

- The **put( ) function** needs to be updated to take into account self.deleted

```python
def put(self,key,data):
    hashcode = self.hashfunction(key)
    if self.slot[hashcode] == None or self.slot[hashcode] == self.deleted:
        self.slot[hashcode] = key
        self.data[hashcode] = data
    elif self.slot[hashcode] == key:
        self.data[hashcode] = data
    else:
        nextslot = self.rehash(hashcode)
        while self.slot[nextslot] != None \
                and self.slot[nextslot] != key:

            nextslot = self.rehash(nextslot)

            if self.slot[nextslot] == None:
                self.slot[nextslot] = key
                self.data[nextslot] = data
            else:
                self.data[nextslot] = data
```

This part has a logical error. Rewrite it. You may utilize self.slot[nextslot] == self.deleted

19

The 'in' and 'len' Operators

- The **__len__( )** allows the use of the len operator.
  The **__contains__( )** allows the use of the in operator.

```python
def __len__(self):
    count = 0
    for value in self.slot:
        if value != None and value != self.deleted:
            count += 1
    return count


def __contains__(self, key):
    return self.get(key) != None
```

Hashing Analysis

- The time complexity of search, insertion, and deletion operations of a hash map is constant time, that is, **O(1)**.

- It must keep the load factor λ small to maintain the time complexity of O(1).

- Hashing provides a useful data structures like hash table or hash map.

학습 정리

1) __setitem__, __getitem__ 을 재정의함으로 대괄호를([ ])를 사용하여 인덱싱할 수 있다

2) 해시맵에서 노드를 삭제하려고 할 때, None과 deleted 값을 가진 노드들을 유의해야한다.