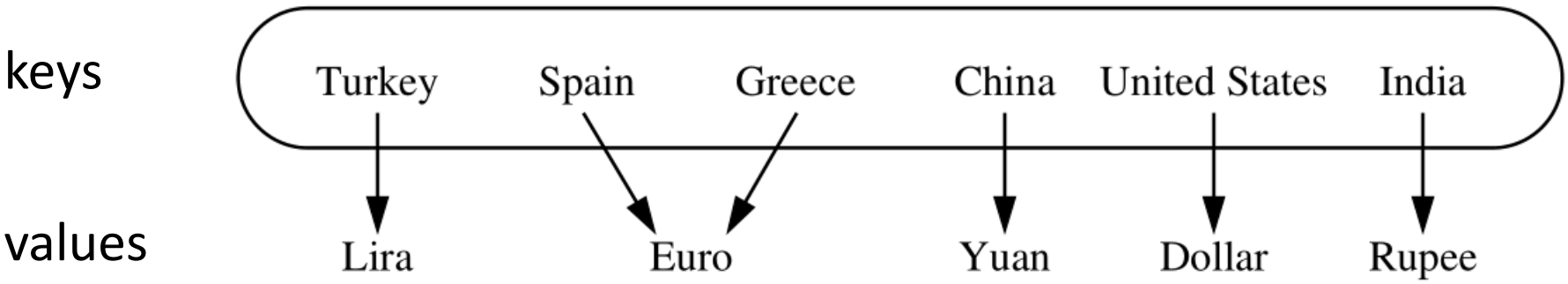
11
Maps and Hash Tables
Chapter 10

Maps

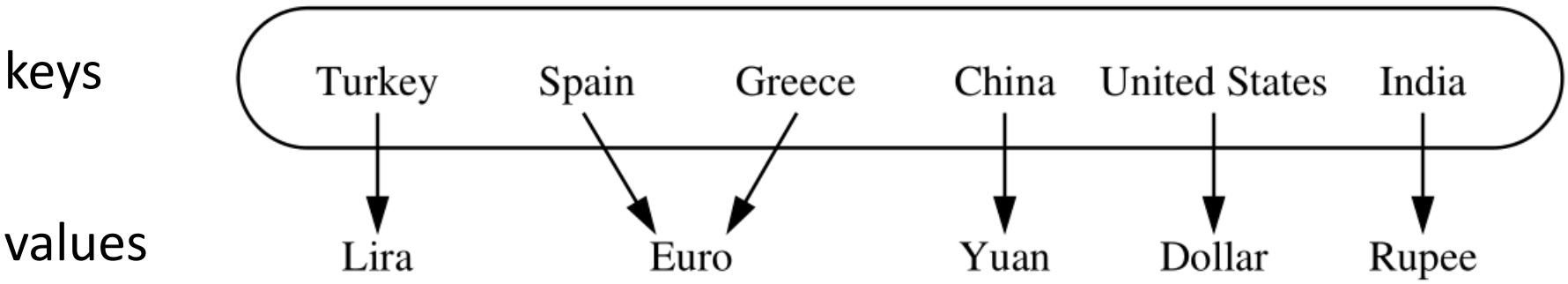


Maps are structures defining associations between keys and values.

Keys have to be distinct.

Maps are also called associative arrays.

Maps



With maps, we can use syntax such as `currency['Turkey']`, and we can change value of a key with `currency['Turkey']= 'Para'`.

Keys do not have to be consecutive values or numerical.

A well-known example in Python is the `dict` class.

The Map ADT

M[k] : Return the value associated with k. If value does not exist raise KeyError (__getitem__).

M[k] = v : Associate the value v with key k. If k already exists, replace its value with v (__setitem__).

del M[k] : Remove the key-value pair (item) whose key is k. If there is no item associated with k, raise a KeyError (__delitem__).

len(M) : Return number of items in the map (__len__).

iter(M) : Returns a generator for the keys in the map (__iter__).

The Map ADT

Some other convenience methods

k in M: Returns true if an item k exists in the map (__contains__).

M.get(k, d=None): Return the value if key k exists, otherwise return default value d.

M.setdefault(k,d): If key k exists, return the value associated with the key, otherwise associate key k with value d, and return d.

M.pop(k, d=None): Return the value associated with key k. If key k does not exist, return d. If d is None, raise exception.

M.popitem(): Return an arbitrary (k,v) tuple and remove it from the map. If map is empty, raise KeyError exception.

The Map ADT

Operation	Return Value	Map
len(M)	0	{ }
M['K'] = 2	–	{'K': 2}
M['B'] = 4	–	{'K': 2, 'B': 4}
M['U'] = 2	–	{'K': 2, 'B': 4, 'U': 2}
M['V'] = 8	–	{'K': 2, 'B': 4, 'U': 2, 'V': 8}
M['K'] = 9	–	{'K': 9, 'B': 4, 'U': 2, 'V': 8}
M['B']	4	{'K': 9, 'B': 4, 'U': 2, 'V': 8}
M['X']	KeyError	{'K': 9, 'B': 4, 'U': 2, 'V': 8}
M.get('F')	None	{'K': 9, 'B': 4, 'U': 2, 'V': 8}
M.get('F', 5)	5	{'K': 9, 'B': 4, 'U': 2, 'V': 8}
M.get('K', 5)	9	{'K': 9, 'B': 4, 'U': 2, 'V': 8}
len(M)	4	{'K': 9, 'B': 4, 'U': 2, 'V': 8}
del M['V']	–	{'K': 9, 'B': 4, 'U': 2}
M.pop('K')	9	{'B': 4, 'U': 2}
M.keys( )	'B', 'U'	{'B': 4, 'U': 2}
M.values( )	4, 2	{'B': 4, 'U': 2}
M.items( )	('B', 4), ('U', 2)	{'B': 4, 'U': 2}
M.setdefault('B', 1)	4	{'B': 4, 'U': 2}
M.setdefault('A', 1)	1	{'A': 1, 'B': 4, 'U': 2}
M.popitem( )	('B', 4)	{'A': 1, 'U': 2}

Application of Map

```
1  freq = { }
2  for piece in open(filename).read( ).lower( ).split( ):
3    # only consider alphabetic characters within this piece
4    word = ''.join(c for c in piece if c.isalpha( ))
5    if word:    # require at least one alphabetic character
6      freq[word] = 1 + freq.get(word, 0)
7
8  max_word = ''
9  max_count = 0
10 for (w,c) in freq.items( ):        # (key, value) tuples represent (word, count)
11   if c > max_count:
12     max_word = w
13     max_count = c
14 print('The most frequent word is', max_word)
15 print('Its number of occurrences is', max_count)
```

Python's MutableMapping ABC

"MutableMapping is an abstract base class (ABC) for generic container for associating key-value pairs."

It is an abstract base class in collections module.

https://hg.python.org/cpython/file/default/Lib/_collections_abc.py

It inherits from Mapping ABC, which is also in the collections module.

Mapping abstract class has implementations for immutable map operations.

MutableMapping abstract class has definitions from the Mapping class and also other mutable operation implementation.

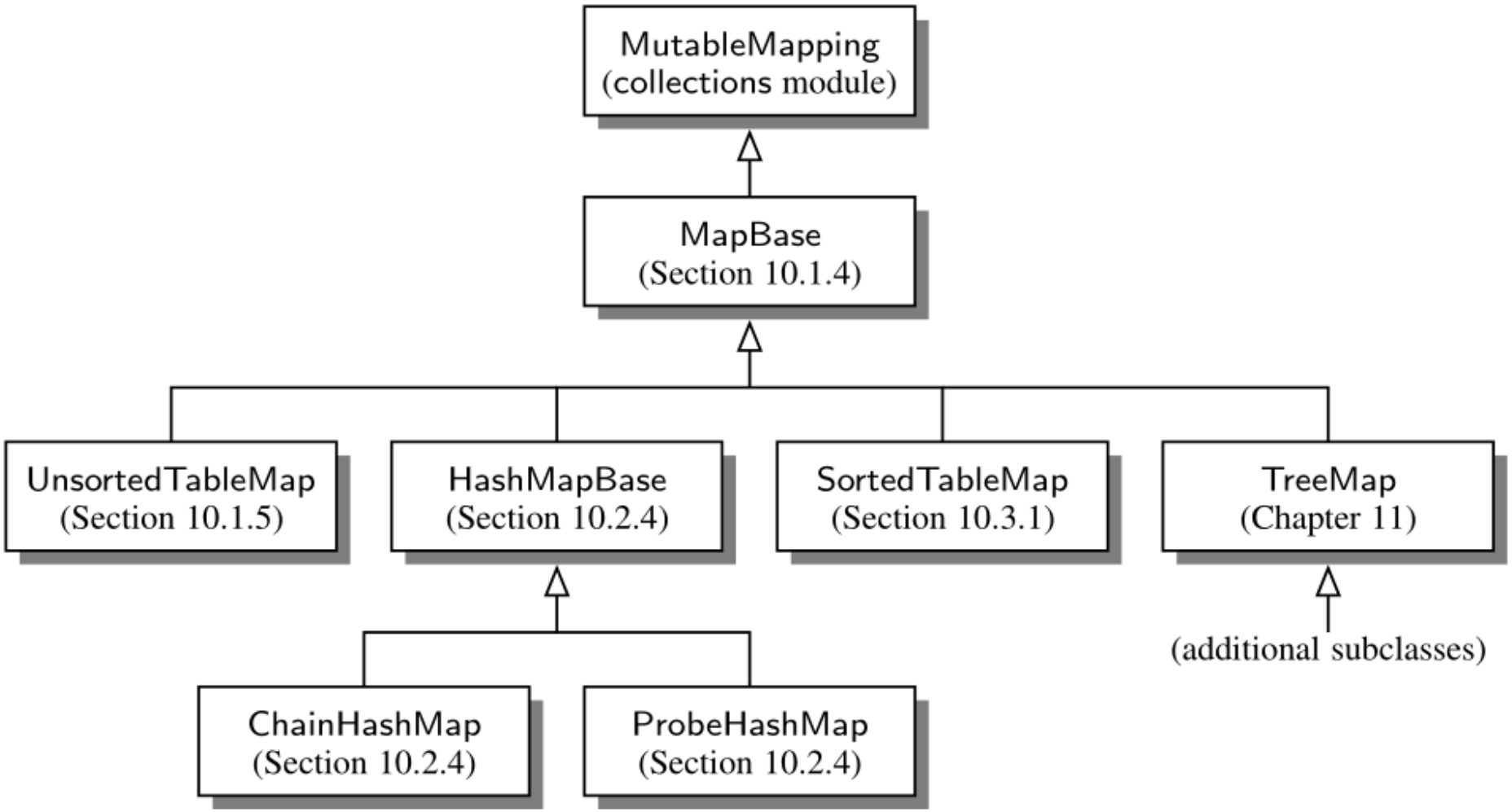
Python's MutableMapping ABC

For mapping, we will have various approaches and, thus, there will be various implementations.

All of these implementations will inherit from a class, MapBase, in which we implement only the _Item nested class, which will represent the "item" that we discussed in ADT operations, which is comprised of a key and value pair.

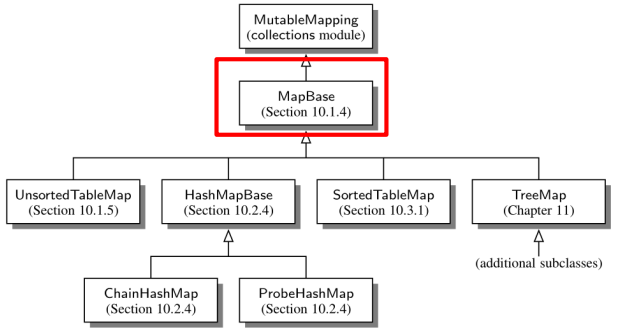
The MapBase class will inherit from MutableMapping. As a result, our MapBase class will have all the methods of MutableMapping class.

Hierarchy of Map Types



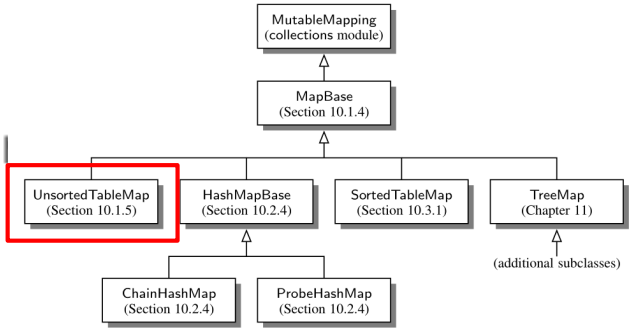
Hierarchy of Map Types

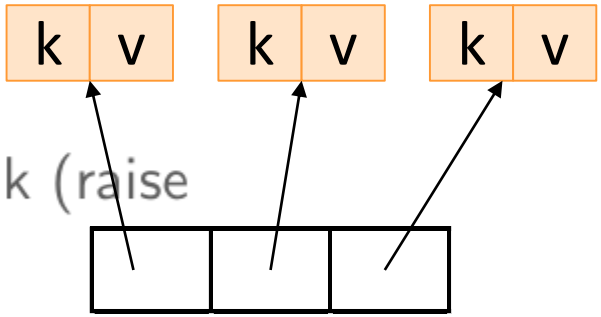
```
1   class MapBase(MutableMapping):
5     class _Item:
7       __slots__ = '_key', '_value'
8
9       def __init__(self, k, v):
10        self._key = k
11        self._value = v
12
13      def __eq__(self, other):
14        return self._key == other._key
15
16      def __ne__(self, other):
17        return not (self == other)
18
19      def __lt__(self, other):
20        return self._key < other._key
```



UnsortedTableMap

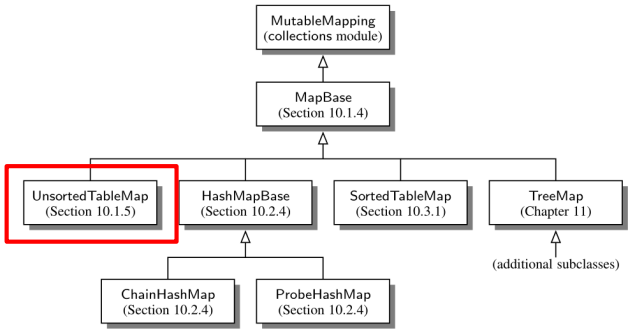
```
1  class UnsortedTableMap(MapBase):
2    """Map implementation using an unorde
3
4    def __init__(self):
5      """Create an empty map."""
6      self._table = [ ]
7
8    def __getitem__(self, k):
9      """Return value associated with key k (raise
10     for item in self._table:
11       if k == item._key:
12         return item._value
13     raise KeyError('Key Error: ' + repr(k))
```

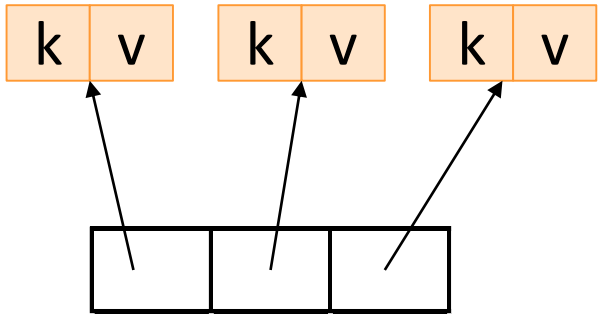




UnsortedTableMap

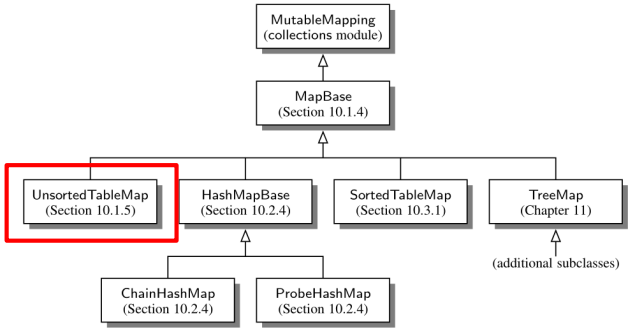
```
15    def __setitem__(self, k, v):
16      """Assign value v to key k, overwriting exist
17      for item in self._table:
18        if k == item._key:
19          item._value = v
20          return
21      # did not find match for key
22      self._table.append(self._Item(k,v))
23
24    def __delitem__(self, k):
25      """Remove item associated with key k (rais
26      for j in range(len(self._table)):
27        if k == self._table[j]._key:
28          self._table.pop(j)
29          return
30      raise KeyError('Key Error: ' + repr(k))
```

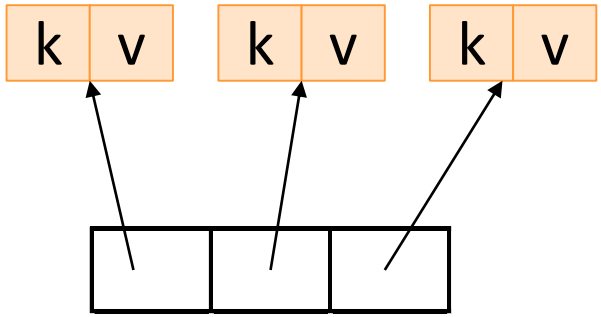




UnsortedTableMap

```
32    def __len__(self):
33        """Return number of ite
34        return len(self._table)
35
36    def __iter__(self):
37        """Generate iteration of
38        for item in self._table:
39            yield item._key
```

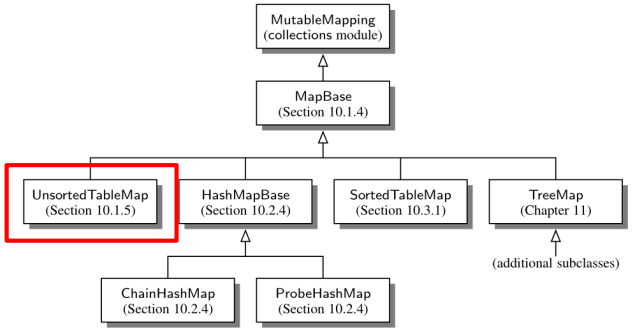


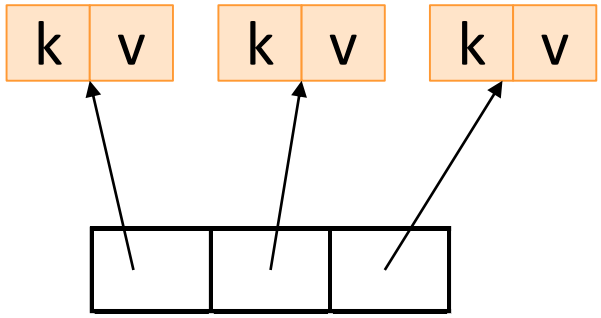


UnsortedTableMap

Get item, delete item, set item operations need to make search on the internal array.

Due to that reason, all of these operations work in O(n) time.

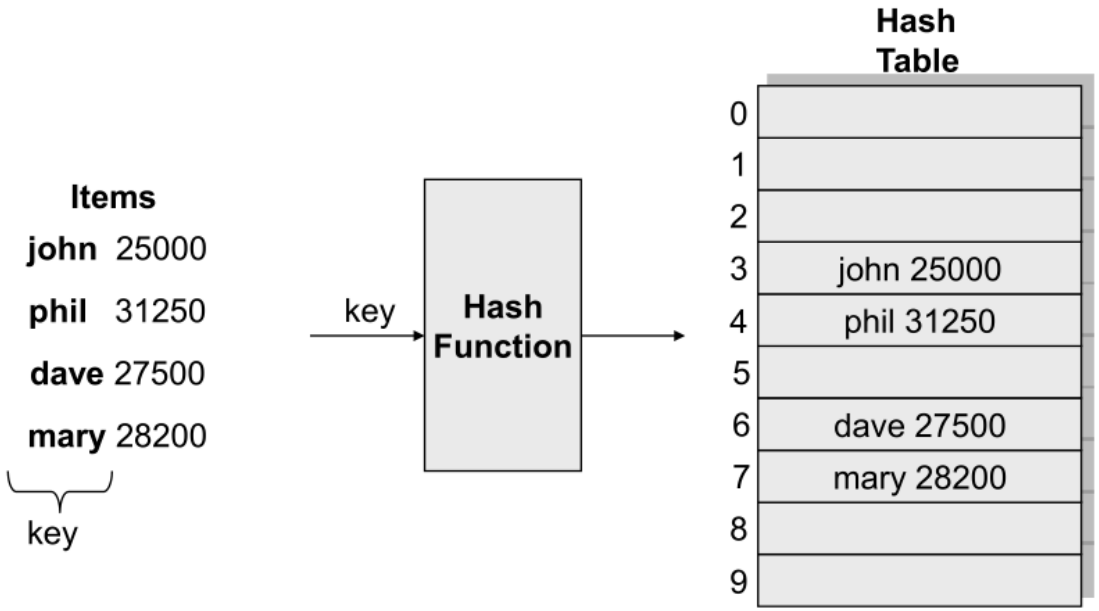




Hash Tables

We can have a map whose get item (`d['A']`), set item (`d['A'] = 5`), and delete item (`del d['A']`) operations can be performed in **O(1)**.

This can be done with a technique called **hashing**.

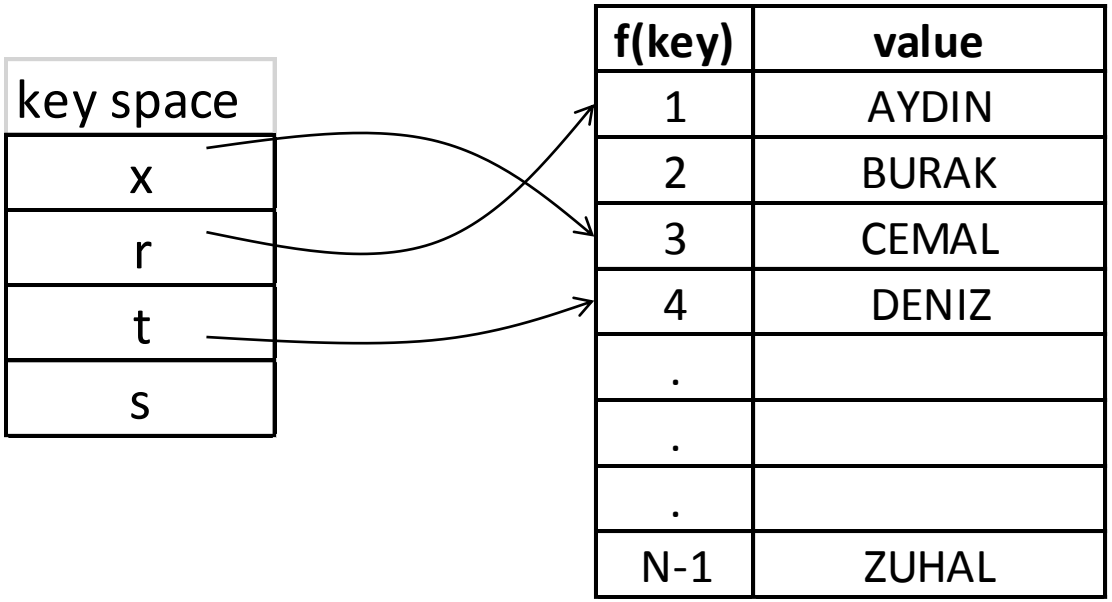


Hash Tables

The hash function must be simple to implement.

It must evenly distribute the keys among the cells in the table.

Let N be the size of the hash table, then the hash function is a function from key space to {0, 1, ..., N-1}.

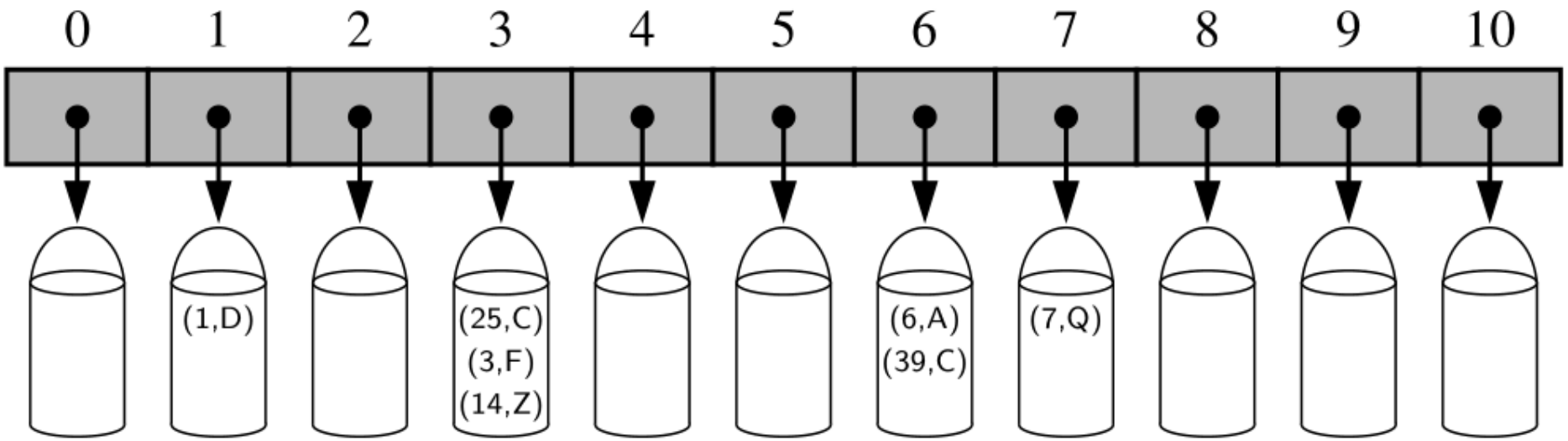


`d['x']` returns `'CEMAL'`

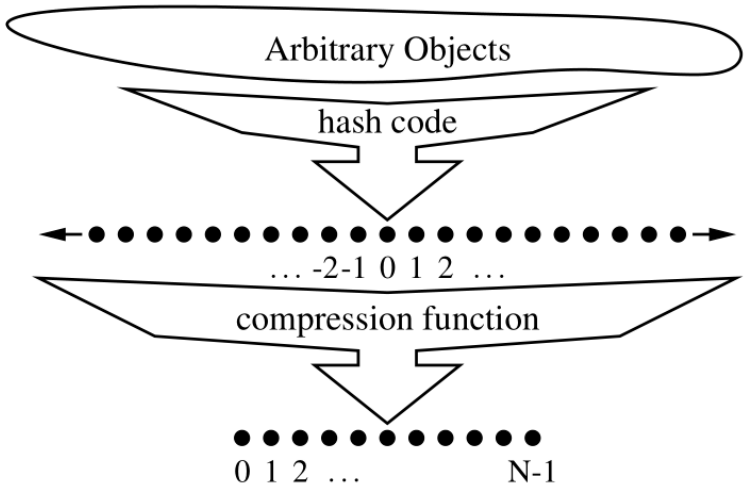
f has to be simple O(1).

Hash Tables

There might be cases where two different keys might map to the same value. Due to that reason, we can use a structure called **bucket array**.



Hash Function



`key --> Z`

`Z --> [0, N-1]`

Separating hash function into two components is a good idea.

We can create hash code for any kind of object regardless of the underlying hash table size.

With the compression part we can create a mapping from hash code to [0, N-1].

Hash Code

Creating a hash code is all about converting an immutable (Why?) object to an integer.

There are various ways to do that.

Hash Code

Using the Bit Representation Object

We can convert any (serializable) object to binary numbers.

```
In [31]: format(16, '032b')
Out[31]: '00000000000000000000000000010000'

In [32]: [format(x, '08b') for x in bytes('ABCD', 'utf-8')]
Out[32]: ['01000001', '01000010', '01000011', '01000100']

In [33]: ''.join([format(x, '08b') for x in bytes('ABCD', 'utf-8')])
Out[33]: '01000001010000100100001101000100'
```

Then we can do 32-bit, 16-bit, 24-bit, etc. hashing.

This means, simply chunk the bit representation into pieces such that each chunk is of X bits.

Hash Code

```
In [33]: ''.join([format(x, '08b') for x in bytes('ABCD', 'utf-8')])
Out[33]: '01000001010000100100001101000100'
```

Here we have two 16-bit chunks. Let's call them X_0 and X_1.

Then we can perform any bitwise operation between X_0 and X_1:

$X_0 + X_1$ or X_0 X-OR X_1.

Eventually what we would get a 16-bit representation which can be easily converted to an integer.

Hash Code

```
In [33]: ''.join([format(x, '08b') for x in bytes('ABCD', 'utf-8')])
Out[33]: '01000001010000100100001101000100'
```

Here we have two 16-bit chunks. Let's call them X_0 and X_1.

Then we can perform any bitwise operation between X_0 and X_1:

$X_0 + X_1$ or X_0 X-OR X_1.

Eventually what we would get a 16-bit representation which can be easily converted to an integer.

However, that can cause to collision for similar objects:

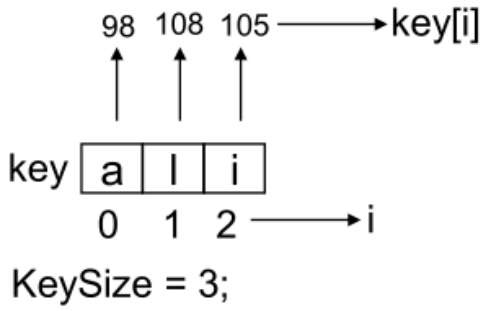
For example, using a 16-bit hash code for words 'stop' and 'tops' would create the same hash code (How?).

Hash Code

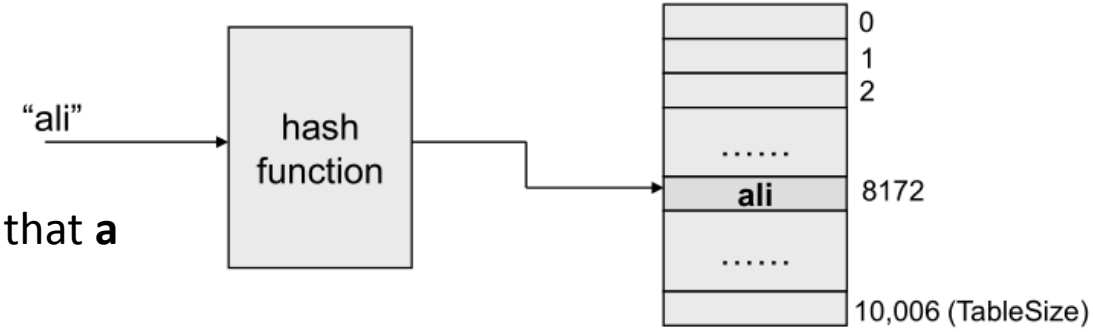
In order to avoid collisions, we can use polynomial hash codes.

For each of the chunk, we can use a multiplier constant:

$$x_0 a^{n-1} + x_1 a^{n-2} + \cdots + x_{n-2} a + x_{n-1}$$



hash("ali") = (105 * 1 + 108*37 + 98*37^2) % 10,007 = 8172



It is recommended that **a** is a prime number.

Hash Code

```
In [33]: ''.join([format(x, '08b') for x in bytes('ABCD', 'utf-8')])
Out[33]: '01000001010000100100001101000100'
```

Bit Shifting

We can also shift bits of X_0 and X_1 prior to add/x-or operation.

00111101100101101010100010101000

10110010110101010001010100000111

```
sum = 0
for each chunk c in chunks:
     sum = c + (bit-wise shifted version of sum)
```

Hash Code

In Python, there is a built-in called `hash` with which we can create hash codes.

It works for immutable data types such as `int`, `float`, `frozensets`, `tuples`, and `strings`.

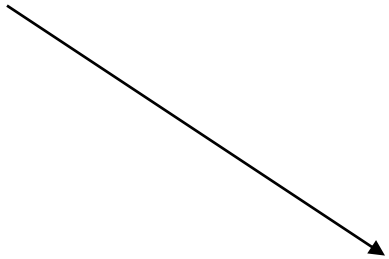
If the operator function `__hash__` is implemented in any class, the `hash` built-in function works. However, if `a` and `b` are objects of a class and `a == b` is `true`, then `hash(a) == hash(b)` should also be `true`.

```
In [50]: class MyClass:
    ...:     a = 1
    ...:     b = 'abc'
    ...:     def __hash__(self):
    ...:
    ...:         return hash((self.a, self.b))
    ...:

In [51]: x = MyClass()

In [52]: hash(x)
Out[52]: 7938775248286252871
```

Compression Function

We can use compression functions to map hash codes to an integer in [0, N-1] where N is the size of the hash table. The result of this operation is an index to the hash table.

Z -- > [0, N-1]



f(key)	value
1	AYDIN
2	BURAK
3	CEMAL
4	DENIZ
.	
.	
.	
N-1	ZUHAL

Compression Function

The Division Method

The index is calculated as follows,

i mod N,

where N is the size of the hash table. N is recommended to be a prime number.

Using a prime number does not guarantee a collision-free hashing, however, decreases the chance to 1/N (e.g., hash codes 2*37+3 and 5*37+3 would collide.).

Compression Function

The MAD Method

MAD is an abbreviation for multiply-add-and-divide.

With this method, a hash code i is mapped to

$[(ai + b) \bmod p] \bmod N$

where

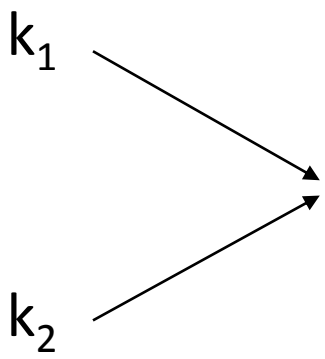
N is the size of the hash table,

p is a prime number larger than N,

a and b are integers chosen from [0, p-1] randomly and $a > 0$.

Collision Handling

Collisions occur when there are two distinct keys (k_1 and k_2) whose hash functions create the same value ($h(k_1) = h(k_2)$).



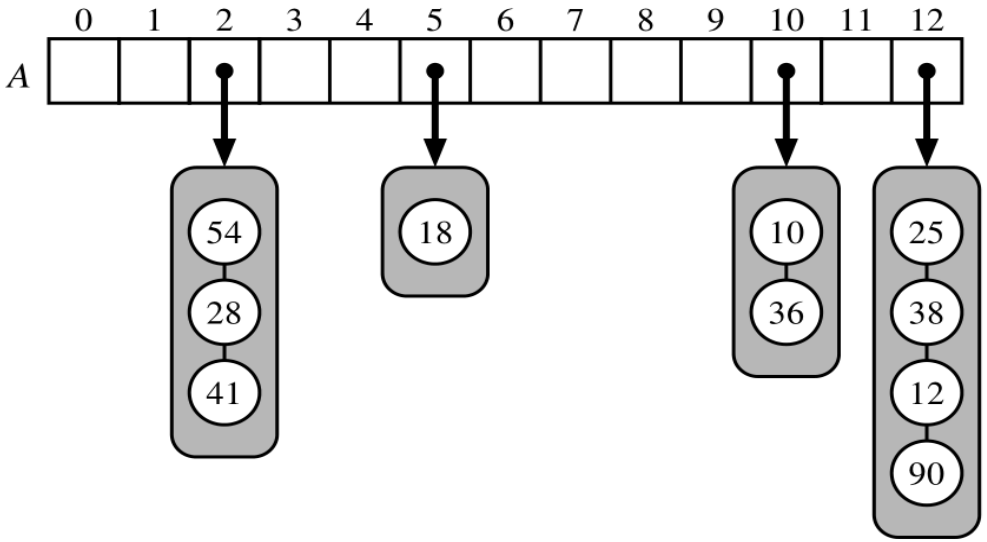
f(key)	value
1	AYDIN
2	BURAK
3	CEMAL
4	DENIZ
.	
.	
.	
N-1	ZUHAL

Collision Handling

Separate Chaining

A hash table is comprised of buckets, and in each bucket there can be a small map instance implemented with a list (an unsorted map).

For each key-value (k,v) pair in a bucket, h(k) would be the same.



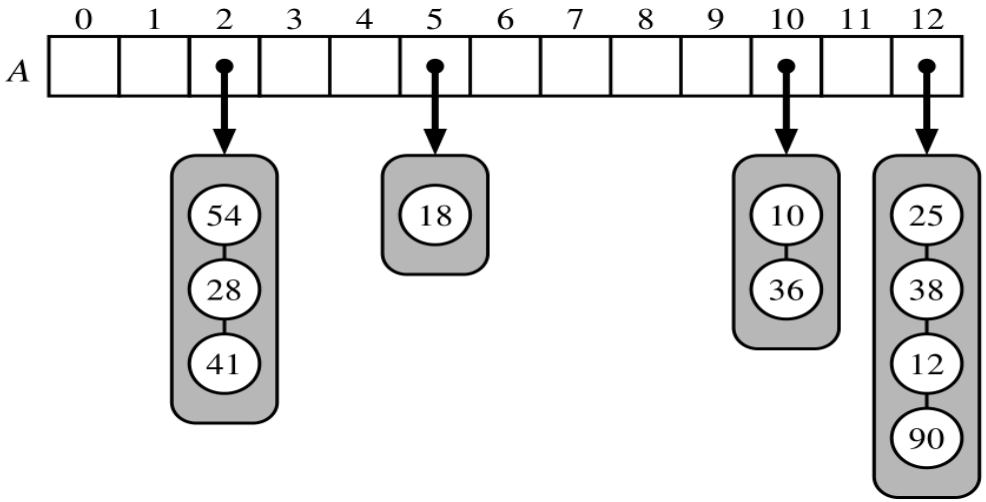
Collision Handling

Separate Chaining

The expected size of a bucket is n/N, where n is the number of (k,v) pairs and N is the size of hash table.

With a good hash function, core map operations run in O(⌈n/N⌉).

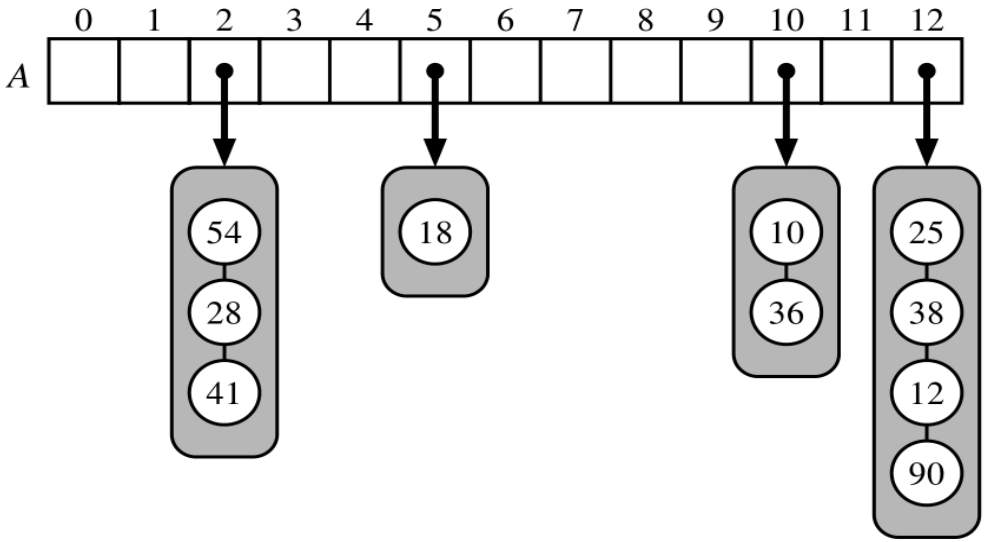
λ = n/N is called the **load factor**.



Collision Handling

Separate Chaining

If λ is O(1), the core map operations run in O(1) expected time.



Collision Handling

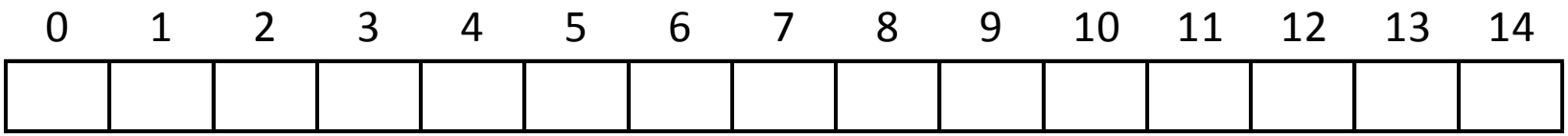
Open Addressing (Probing)

Instead of a bucket array, we use an ordinary array to store (k,v) pairs.

To find the array index to insert a new item,

(h(k) + f(i)) mod N

is used, where i >= 0 is the number of trials until an empty slot is found.



Collision Handling

Linear Probing

f(i) = i, so the new item (k,v) will be inserted to

A[(h(k)+0) mod N] in the 1st trial,

A[(h(k)+1) mod N] in the 2nd trial, and

A[(h(k)+i) mod N] in the (i+1)st trial.

N=15

0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	
				19											Insert 19, i=0
				19	34										Insert 34, i=1
				19	34	49	20								Insert 49, i=2 Insert 20, i=2

Collision Handling

Linear Probing

When searching for a key, we start with A[h(k) mod N], if unsuccessful continue with A[(h(k)+1) mod N], so on so forth, until an space is reached.

For example, for the hash table below, a search for 64 will fail at index 8 (i=4), and another one for 25 will immediately fail at 10 (i=0).

0	1	2	3	4	5	6	7	8	9	10	11	12	13	14
				19	34	49	20							

Collision Handling

Linear Probing

For deletion operations, an "available" marker can be used.

For example, for the hash table below, after deleting 49, a search for 20 will be successful only if we mark the deleted cell as available.

0	1	2	3	4	5	6	7	8	9	10	11	12	13	14
				19	34	49	20							

0	1	2	3	4	5	6	7	8	9	10	11	12	13	14
				19	34	av	20							

Collision Handling

Linear Probing

The problem with the linear probing, clustering in the hash table array can develop quickly.

Long clusters can adversely affect the run time performance.

0	1	2	3	4	5	6	7	8	9	10	11	12	13	14
				19	34	49	64	79	94	109				

We need to do 7 probing operations for key 109.

Collision Handling

Quadratic Probing

f(i) = i^2, so the new item (k,v) will be inserted to

A[(h(k)+0) mod N] in the 1st trial,

A[(h(k)+1) mod N] in the 2nd trial,

A[(h(k)+4) mod N] in the 3rd trial, and

A[(h(k)+i^2) mod N] in the (i+1)st trial.

N=15

0	1	2	3	4	5	6	7	8	9	10	11	12	13	14
				19										

Insert 19, i=0

0	1	2	3	4	5	6	7	8	9	10	11	12	13	14
				19	34									

Insert 34, i=1

0	1	2	3	4	5	6	7	8	9	10	11	12	13	14
				19	34			49						

Insert 49, i=2

Collision Handling

Quadratic Probing

If N is a prime number and the array is less than half full, then quadratic probing is guaranteed to find an empty slot.

Otherwise, finding a space is not guaranteed.

Collision Handling

Probing with Double Hashing

f(i) = i*h'(k), so the new item (k,v) will be inserted to

A[(h(k)+0) mod N] in the 1st trial,

A[(h(k)+h'(k)) mod N] in the 2nd trial, and

A[(h(k)+i*h'(k)) mod N] in the (i+1)st trial.

A common choice for h' is,

h'(k) = q - (k mod q)

where q is a prime number and q < N, and N is a prime number.

Collision Handling

Probing with Pseudo Random Number Generator

f(i) is a generator that generates (the same) pseuod random numbers.

This is the approach used by

Python's dictionary class.

```
In [69]: from random import random

In [70]: from random import seed

In [71]: seed(1)

In [72]: int(random()*1000)
Out[72]: 134

In [73]: int(random()*1000)
Out[73]: 847

In [74]: int(random()*1000)
Out[74]: 763

In [75]: seed(1)

In [76]: int(random()*1000)
Out[76]: 134

In [77]: int(random()*1000)
Out[77]: 847

In [78]: int(random()*1000)
Out[78]: 763
```

Performance of Hash Tables

Operation	List	Hash Table	
		expected	worst case
__getitem__	$O(n)$	$O(1)$	$O(n)$
__setitem__	$O(n)$	$O(1)$	$O(n)$
__delitem__	$O(n)$	$O(1)$	$O(n)$
__len__	$O(1)$	$O(1)$	$O(1)$
__iter__	$O(n)$	$O(n)$	$O(n)$

Hash Table Implementation

Design Decisions

Bucket array is represented as a Python list.

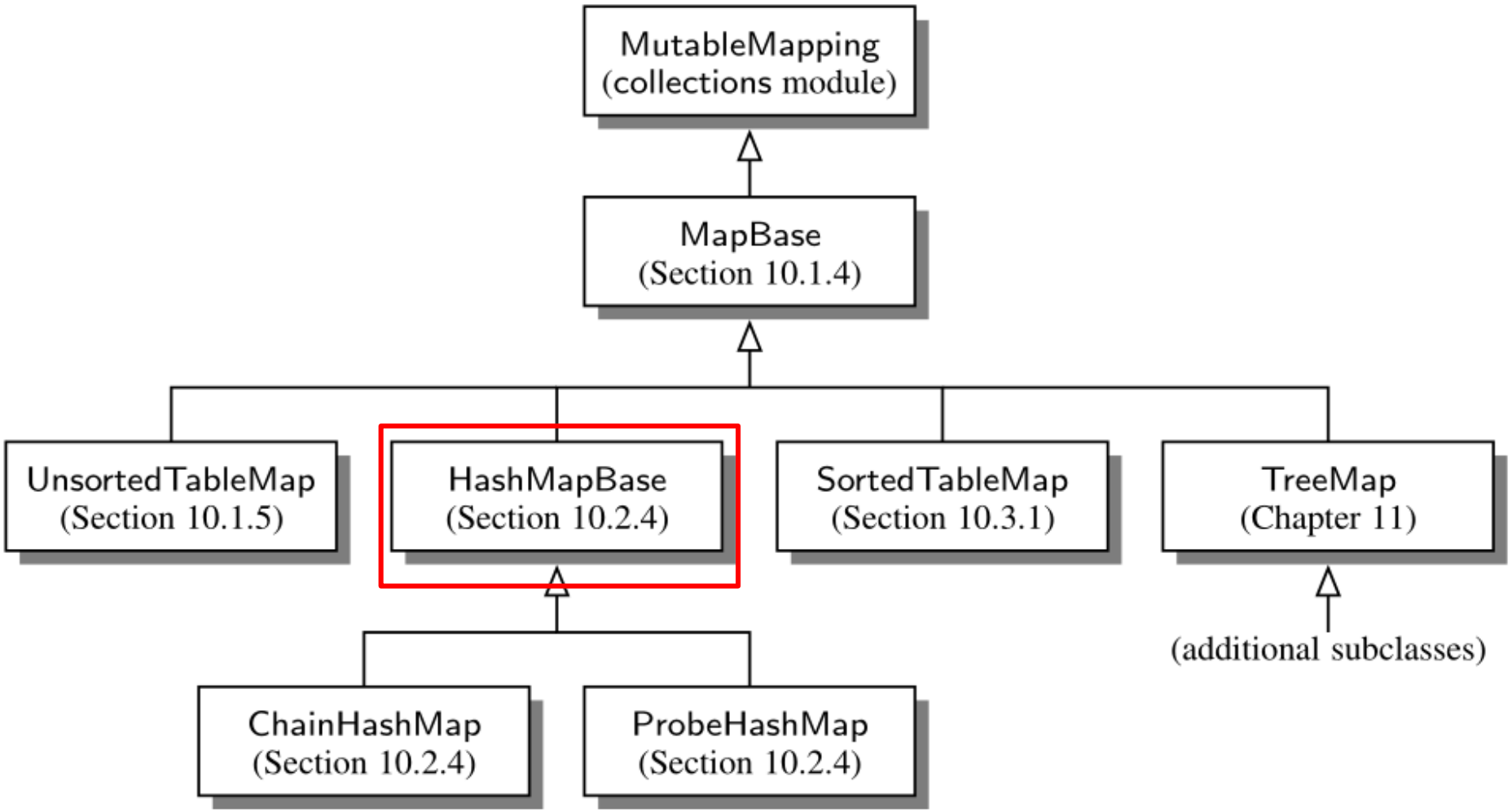
For chaining, each cell in bucket array there will be an instance of UnsortedTableMap instance.

For probing, each cell in bucket array will have a reference to the "value" associated with the "key."

If the load factor is larger than 0.5, the size of the bucket array is doubled.

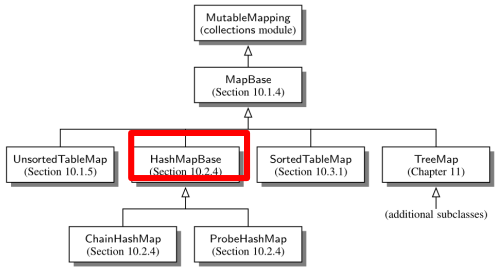
Hash codes are generated with Python's built-in hash method, and as the compression technique a randomized MAD is used.

Hierarchy of Map Types



HashMapBase

HashMapBase is base class for common operations that exist in both ChainHashMap and ProbeHashMap.



It assumes that the following methods are implemented by the child classes:

_bucket_getitem(j, k)

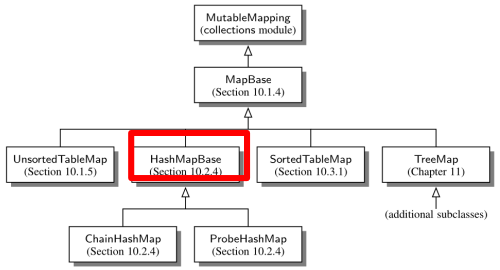
_bucket_setitem(j, k, v)

_bucket_delitem(j, k)

__iter__ (iterate through all keys of the map)

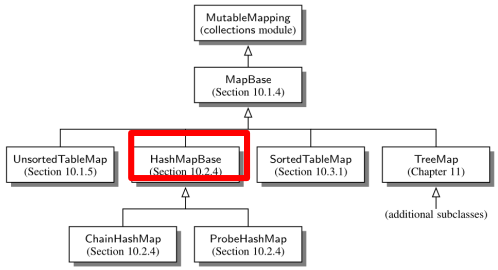
HashMapBase

```
1  class HashMapBase(MapBase):
2    """Abstract base class for map using hash-table with MAI
3
4    def __init__(self, cap=11, p=109345121):
5      """Create an empty hash-table map."""
6      self._table = cap * [ None ]
7      self._n = 0                          # number of entries in the map
8      self._prime = p                      # prime for MAD compression
9      self._scale = 1 + randrange(p-1)     # scale from 1 to p-1 for MAD
10     self._shift = randrange(p)           # shift from 0 to p-1 for MAD
11
12   def _hash_function(self, k):
13     return (hash(k)*self._scale + self._shift) % self._prime % len(self._table)
```



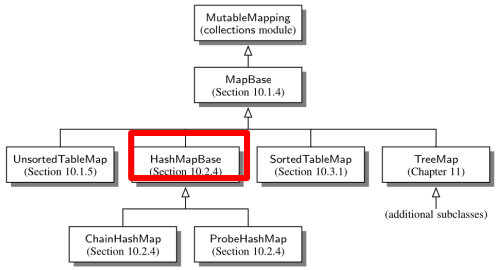
MAD: [(a* hash(k) + b) mod p] mod N

HashMapBase



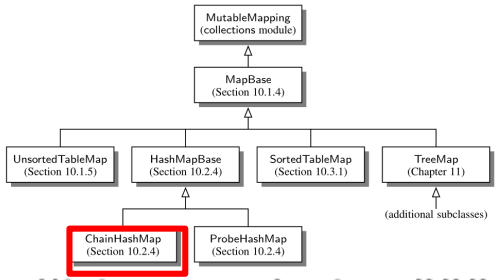

```
15    def __len__(self):
16      return self._n
17
18    def __getitem__(self, k):
19      j = self._hash_function(k)
20      return self._bucket_getitem(j, k)          # may raise KeyError
21
22    def __setitem__(self, k, v):
23      j = self._hash_function(k)
24      self._bucket_setitem(j, k, v)              # subroutine maintains self._n
25      if self._n > len(self._table) // 2:        # keep load factor <= 0.5
26        self._resize(2 * len(self._table) - 1)   # number 2^x - 1 is often prime
```

HashMapBase

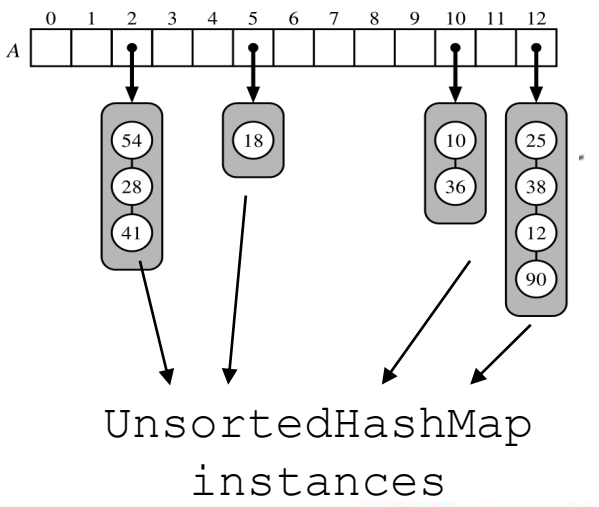



```
28    def __delitem__(self, k):
29      j = self._hash_function(k)
30      self._bucket_delitem(j, k)          # may raise KeyError
31      self._n -= 1
32
33    def _resize(self, c):                 # resize bucket array to capacity c
34      old = list(self.items( ))           # use iteration to record existing items
35      self._table = c * [None]            # then reset table to desired capacity
36      self._n = 0                         # n recomputed during subsequent adds
37      for (k,v) in old:
38        self[k] = v                       # reinsert old key-value pair
```

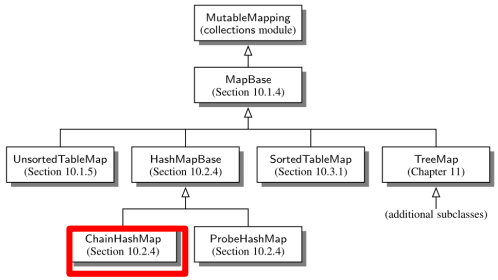
ChainHashMap



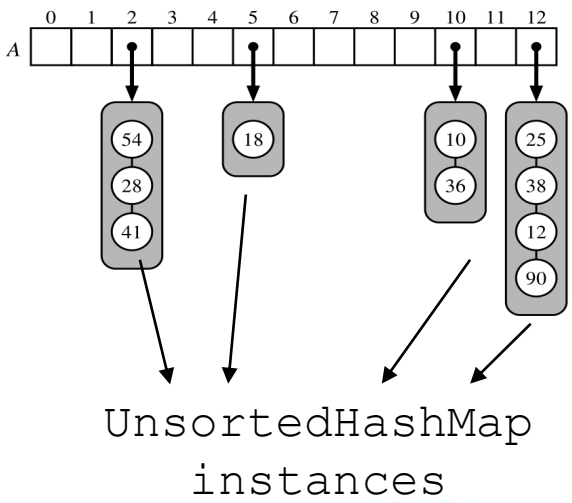

```
1  class ChainHashMap(HashMapBase):
2    """Hash map implemented with separate chaining for collision resolution."""
3
4    def _bucket_getitem(self, j, k):
5      bucket = self._table[j]
6      if bucket is None:
7        raise KeyError('Key Error: ' + repr(k))
8      return bucket[k]
```



ChainHashMap

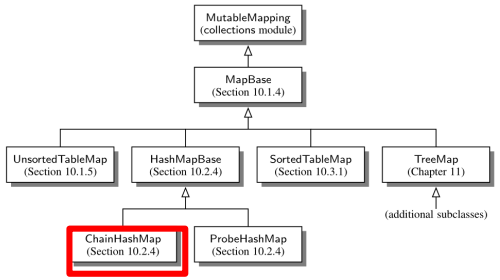


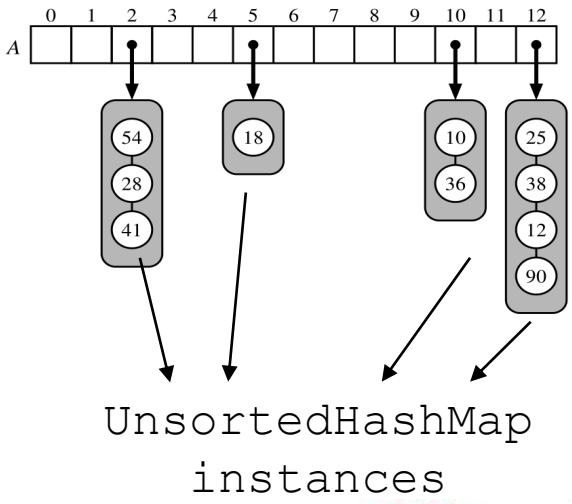

```
10    def _bucket_setitem(self, j, k, v):
11      if self._table[j] is None:
12        self._table[j] = UnsortedTableMap( )
13      oldsize = len(self._table[j])
14      self._table[j][k] = v
15      if len(self._table[j]) > oldsize:
16        self._n += 1
```



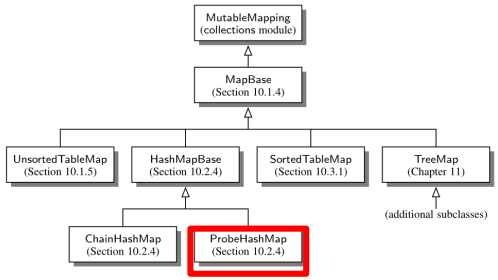
ChainHashMap

```
18    def _bucket_delitem(self, j, k):
19      bucket = self._table[j]
20      if bucket is None:
21        raise KeyError('Key Error: ' + repr(k))
22      del bucket[k]
23
24    def __iter__(self):
25      for bucket in self._table:
26        if bucket is not None:
27          for key in bucket:
28            yield key
```





ProbeHashMap



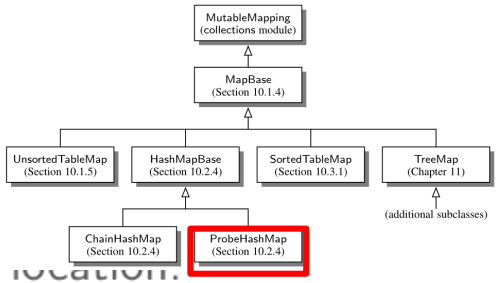

```
1  class ProbeHashMap(HashMapBase):
2    """Hash map implemented with linear probing for collision resolution."""
3    _AVAIL = object( )        # sentinal marks locations of previous deletions
4
5    def _is_available(self, j):
6      """Return True if index j is available in table."""
7      return self._table[j] is None or self._table[j] is ProbeHashMap._AVAIL
```

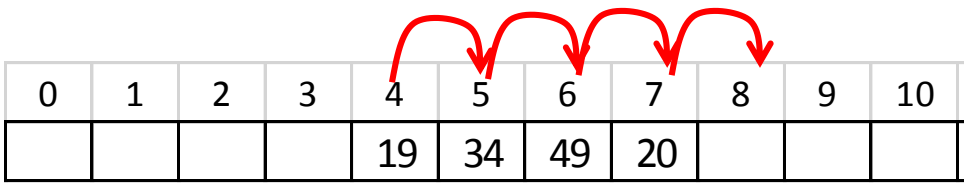
0	1	2	3	4	5	6	7	8	9
				19	34	49	20		

0	1	2	3	4	5	6	7	8	9
				19	34	av	20		

ProbeHashMap

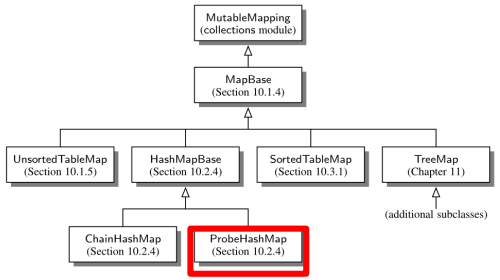
```
9   def _find_slot(self, j, k):
10    """Search for key k in bucket at index j.
11
12    Return (success, index) tuple, described as follows:
13    If match was found, success is True and index denotes its location.
14    If no match found, success is False and index denotes first available slot.
15    """
16    firstAvail = None
17    while True:
18      if self._is_available(j):
19        if firstAvail is None:
20          firstAvail = j                    # mark this as first avail
21        if self._table[j] is None:
22          return (False, firstAvail)        # search has failed
23      elif k == self._table[j]._key:
24        return (True, j)                    # found a match
25      j = (j + 1) % len(self._table)        # keep looking (cyclically)
```





ProbeHashMap

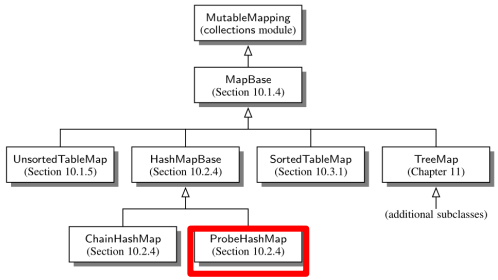
```
26    def _bucket_getitem(self, j, k):
27      found, s = self._find_slot(j, k)
28      if not found:
29        raise KeyError('Key Error: ' + repr(k))
30      return self._table[s]._value
31
32    def _bucket_setitem(self, j, k, v):
33      found, s = self._find_slot(j, k)
34      if not found:
35        self._table[s] = self._Item(k,v)
36        self._n += 1
37      else:
38        self._table[s]._value = v
```



0	1	2	3	4	5	6	7	8
				19	34	49	20	

ProbeHashMap

```
40    def _bucket_delitem(self, j, k):
41      found, s = self._find_slot(j, k)
42      if not found:
43        raise KeyError('Key Error: ' + repr(k))
44      self._table[s] = ProbeHashMap._AVAIL
45
46    def __iter__(self):
47      for j in range(len(self._table)):
48        if not self._is_available(j):
49          yield self._table[j]._key
```



0	1	2	3	4	5	6	7	8
				19	34	49	20	