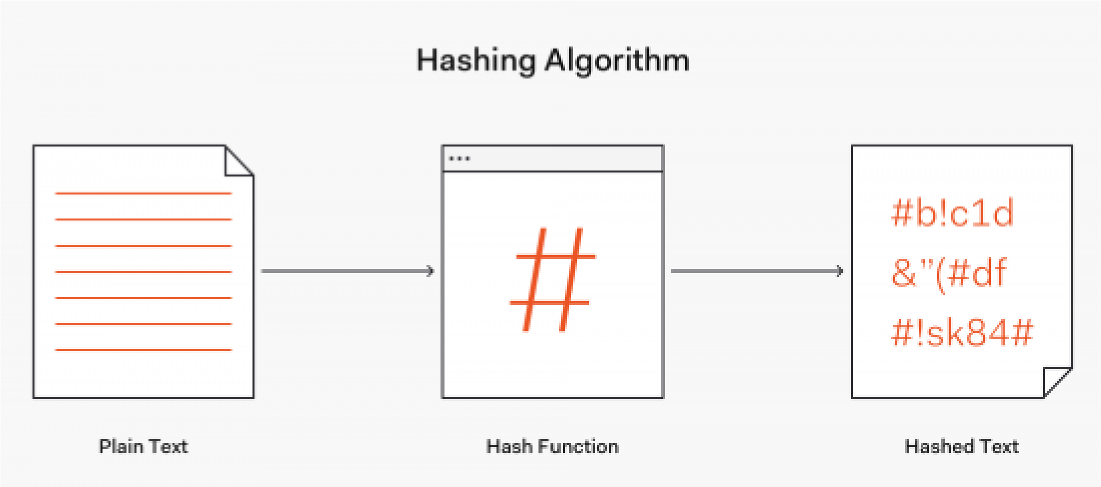
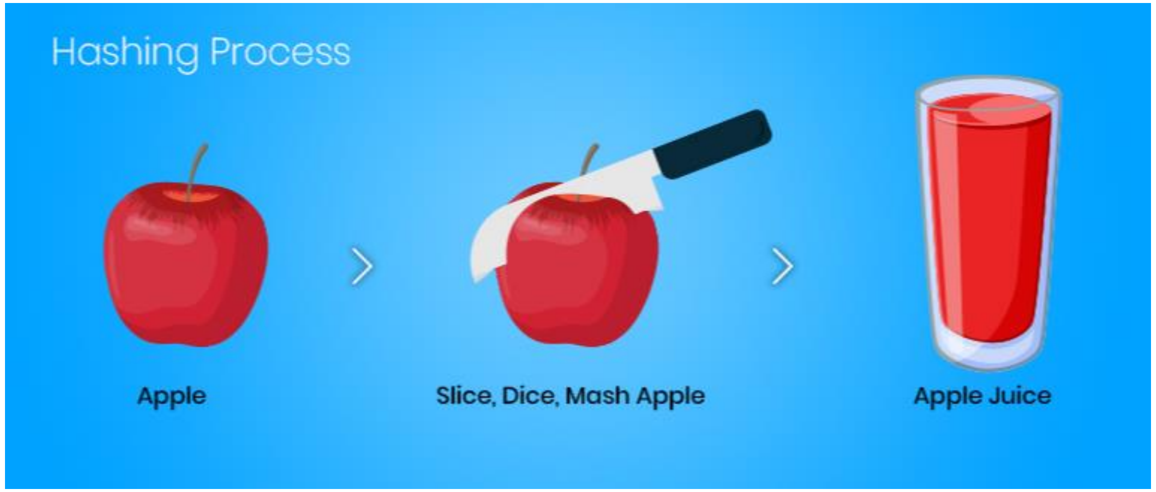
MAPS AND HASH TABLES

School of Artificial Intelligence

PREVIOUSLY ON DS&A

- Maps/dictionaries
- Hash Tables
- Hash Functions
  - Hashing
  - Compressing





MAPS AND DICTIONARIES

- Key->Value pairing
- Keys are (assumed) to be unique, values are not necessarily unique
- Retrieve value with key using m[k]
- Map implementation
  - Unsorted: with underlying array
- Hash function
  - Hashing: produce an integer based on the key
  - Compressing: compress the hash code to fit into [0, N-1], N = capacity of underlying array

HASH CODES（哈希码）

- Hashing: treating the bit representation as an integer
- For any data type X, its representation in memory can be considered an integer
  - For integer 314, h(314) = 314
  - For floating-point number 3.14, h(3.14) will use its memory representation as an integer
- For type that uses longer than a desired hash code
  - E.g. if we want a 32-bit hash code, if a floating-point number uses a 64-bit representation
  - Approaches: take the first/last 32 bit; add the first/last 32 bit, take exclusive-or of the first/last 32 bits

POLYNOMIAL HASH CODES（多项式哈希码）

- Summation and exclusive-or: NOT good choices for character strings or other variable-length objects that can be viewed as tuples of the form $(x_0, x_1, \ldots, x_{n-1})$, where the order of x is significant.
  - E.g. 16-bit hash code for a character string s that sums the Unicode values of the characters in s.
  - "temp01" and "temp10" produces the same hash code.
  - "stop", "tops", "pots", and "spot" produces the same hash code
- A more complicated hashing function is needed, such as

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

- This hash code is called a polynomial hash code

CYCLIC SHIFT HASH CODES

- Replaces multiplication by a with a cyclic shift of a partial sum by a certain number of bits
- 5-bit cyclic shift:
- 00111101100101101010100010101000 to 10110010110101010001010100000111
- Table: comparison of collision behavior for the cyclic-shift hash code to a list of 230,000 English words

```
def hash_code(s):
  mask = (1 << 32) − 1
  h = 0
  for character in s:
    h = (h << 5 & mask) | (h >> 27)
    h += ord(character)
  return h
```

	Collisions	
Shift	Total	Max
0	234735	623
1	165076	43
2	38471	13
3	7174	5
4	1379	3
5	190	3
6	502	2
7	560	2
8	5546	4
9	393	3
10	5194	5
11	11559	5
12	822	2
13	900	4
14	2001	4
15	19251	8
16	211781	37

COMPRESSION FUNCTIONS

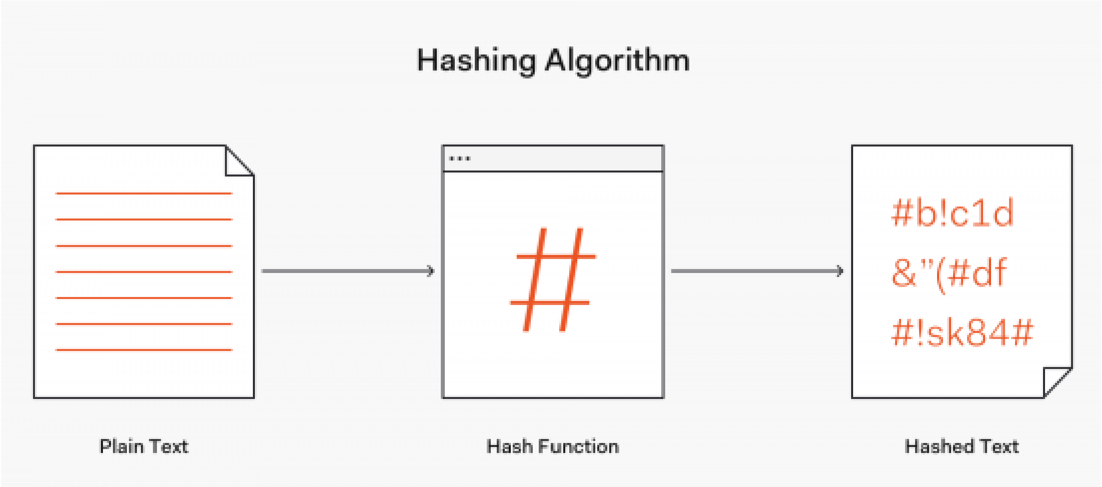
- The hash code for a key *k* may not be suitable for use in a bucket array
- It be negative or may exceed the capacity of the bucket array
- Therefore an additional computation is needed to map the integer into the range [0, N-1] – the **compression function**
- Division method
  - **i mod N**, N = size of the bucket array
  - Choice of N: often prefer prime numbers
  - {200, 205, 210, 215, 220, …, 600} into a bucket array of size 100
  - {200, 205, 210, 215, 220, …, 600} into a bucket array of size 101

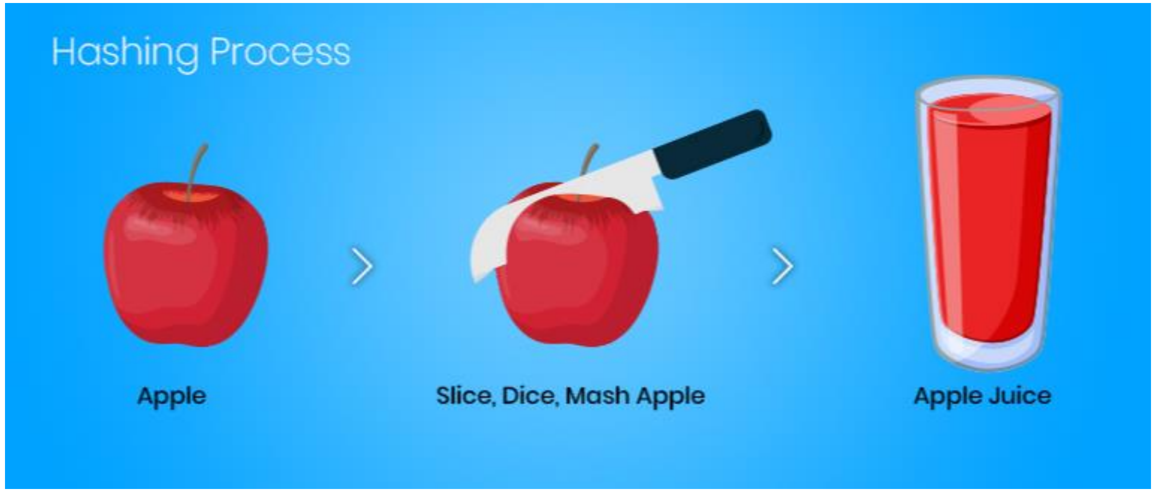
COMPRESSION FUNCTIONS

- MAD (Multiply Add and Divide) method
  - **((ai+b) mod p) mod N**, N = size of the bucket array, p is a prime number larger than N, a and b are integers chosen at random from [0, p-1], with a > 0
  - Finding p: in polynomial time
  - Worst case keys k1 != k2, Pr (h(k1) == h(k2)) = 1/N
- Multiplication method
  - h(k) = ((a.k) mod 2^w) >> (w-r), w = w bits computer, bucket array size N = 2^r
  - A better be odd, and should not be close to powers of 2

THIS LECTURE

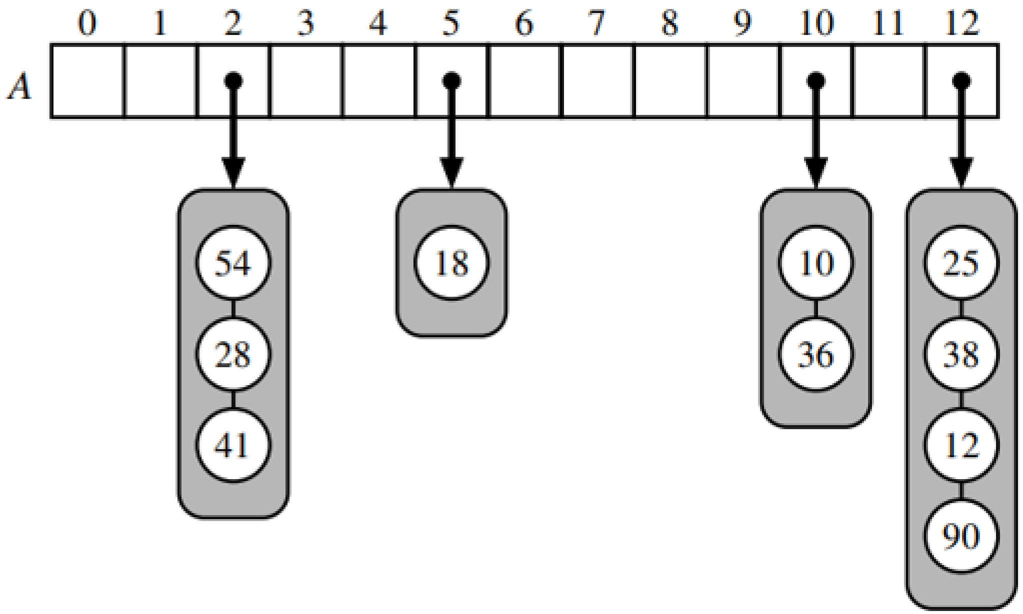
- Collision handling
  - Open addressing
- Efficiency of Hash Tables
- Python Hash Table Implementation
- Separate chaining
- Sorted maps
- Application of sorted maps





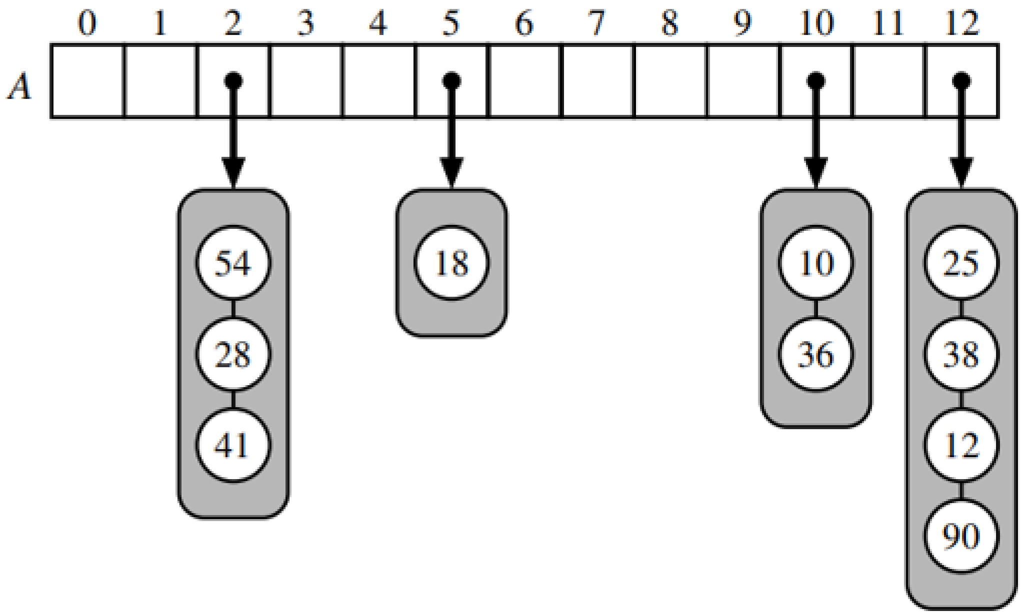
COLLISION HANDLING

- Hash table
  - A bucket array A
  - A hash function h
  - Map to store (k, v) items in the bucket A[h(k)]
- Problem: two distinct keys, k1 and k2, such that h(k1) = h (k2) (hash collision)
- Solution: secondary container at each index (bucket)
- This is often referred to as **separate chaining** （分离链表）



COLLISION HANDLING

- Separate chaining
- Operations on an individual index (bucket) take time proportional to the size of the bucket
- "Good" hash function:
  - expected size of a bucket: n/N
  - n = number of items in the map
  - N = capacity of the bucket array
  - Core map operations run in O(ceiling(n/N))
- Load factor（负载因子）: λ = n/N
- When λ is O(1), operations on the hash table run in O(1)

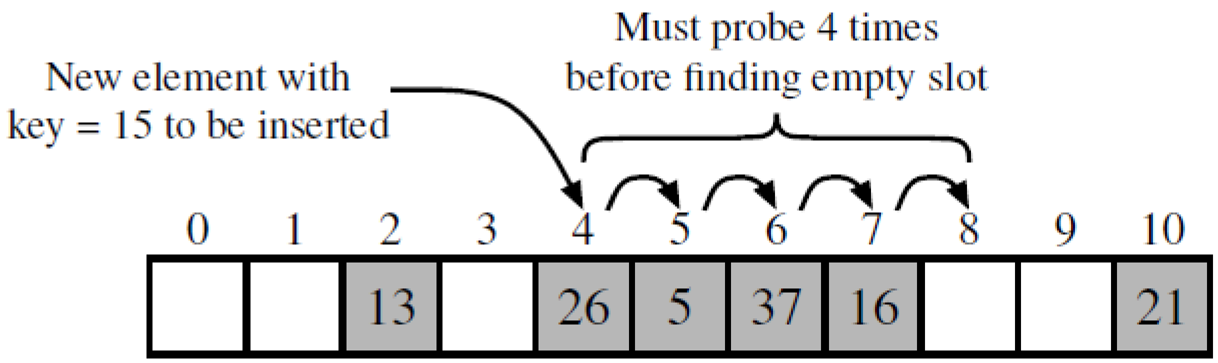


COLLISION HANDLING

- Separate chaining （分离链表）:
  - Advantage: simple implementation
  - Disadvantage: relies on auxiliary data structure - list to hold items with colliding keys
- Alternative approach: open addressing （开放寻址）
  - Always store each item directly in a table slot
  - No auxiliary structures are used
  - Load factor is ways at most 1 and items are stored directly in the cells of the bucket array

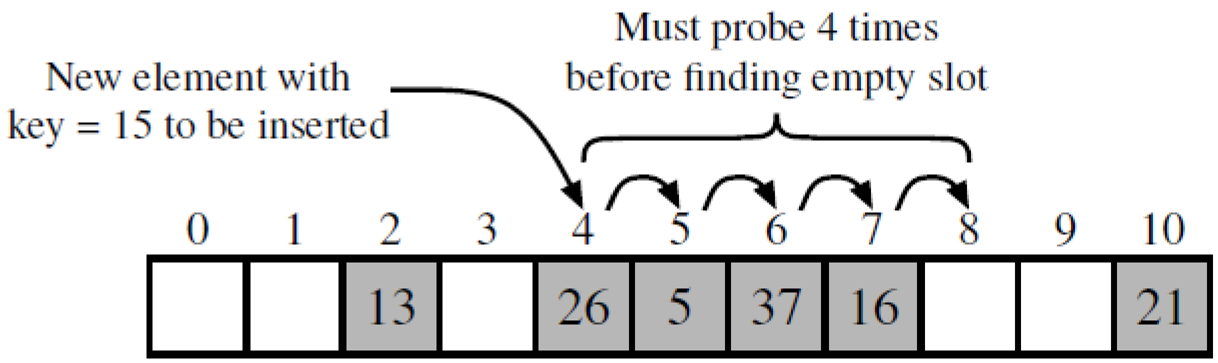
LINEAR PROBING （线性探索）

- Linear probing
- Insert an item (k, v) into a bucket A[j]
- If a[j] is occupied, j = h(k), then try A[(j+1) mod N]
- If A[(j+1) mod N] is occupied, try A[(j+2) mod N], so on
- Need to change the implementation of funcgtions such as __getitem__, __setitem__, __delitem__



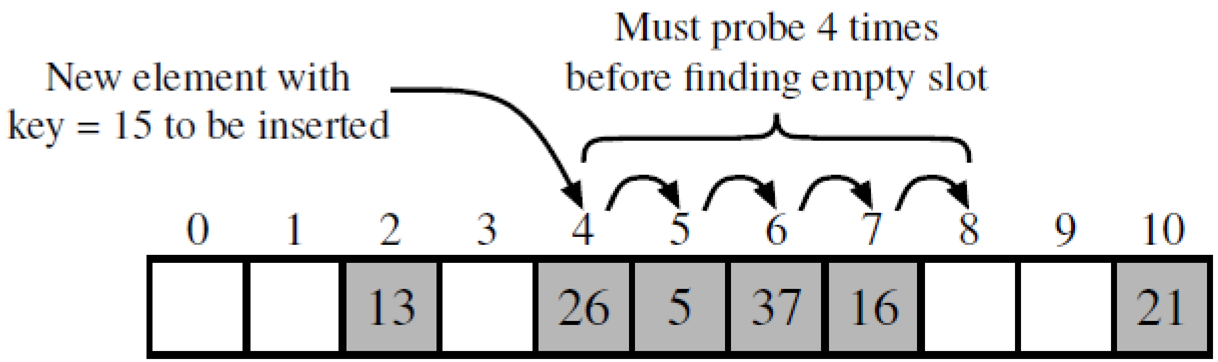
LINEAR PROBING （线性探索）

- Deletion – slightly more complicated
- After insertion for item k=15 deleting 37 (A[6]), a search for k=15 fails
- Probing 4, 5, 6 – empty cell
- Solution: place a ‘available’ marker at A[6], so search for k=15 will pass A[6] until desired item (or empty)
- __setitem__ should remember an available cell, so that a new item (k, v) can be placed there



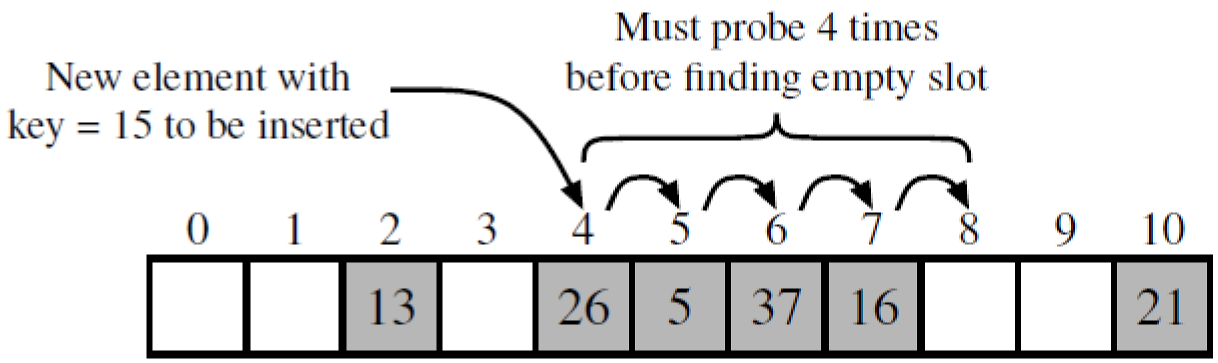
LINEAR PROBING （线性探索）

- Disadvantage?
- Probing takes time proportional to N
- Complicated implementation
- Clustering （聚集）: items tend to stick together due to the probing algorithm



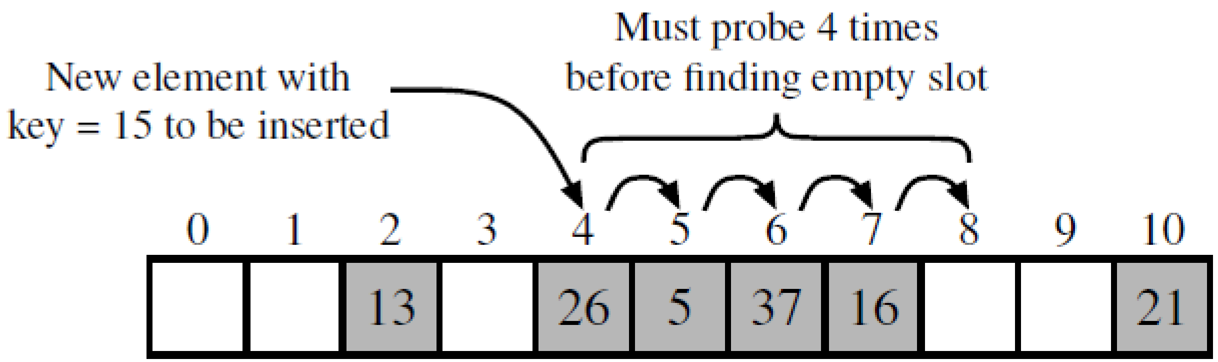
QUADRATIC PROBING （二次探索）

- Iteratively tries A[h(k) + f(i) mod N] for I = 0, 1, 2, …, $f(i) = i^2$
- Spreads the probing distance over the length N
- Deletion – same strategy as linear probing
- Problems again: secondary clustering （二次聚集）
- When the bucket array is half full, or N is not a prime, quadratic probing does not guarantee to find an empty slot



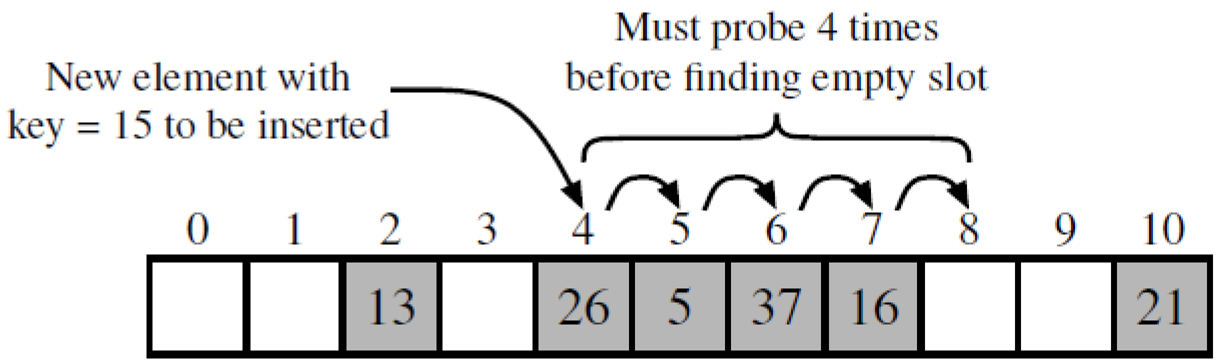
DOUBLE HASHING （二次哈希）

- Secondary hash function h'
- If h maps some key k to a bucket A[h(k)] that is occupied, try A[h(k) + f(i) mod N], for i = 1,2,3, ...
- f(i) = i*h'(k)
- h'(k) cannot be 0
- h'(k) = q – (k mod q), q, N are prime numbers and q < N



ADDITIONAL OPEN ADDRESSING

- A[h(k) + f(i) mod N] where f(i) produces a pseudo-random number
- Repeatable random number



LOAD FACTORS AND REHASING

- Load factor（负载因子）: $\lambda = n/N$ should be kept below 1
- Separate chaining: λ->1 the probability of a collision increases greatly
  - $\lambda < 0.9$ for separate chaining
- Open addressing
  - When λ grows beyond 0.5 and approaches 1, clusters start to show
  - Linear probing: $\lambda < 0.5$
  - Other open addressing: $\lambda < 2/3$
- What happens if an insertion causes the load factor to go beyond 0.5 for linear probing and 2/3 for other open addressing means?
- **Rehashing**: resize the table + reinsert all objects into new table
  - Resize: how?
  - new compression function -> why?

EFFICIENCY OF HASH TABLES

- “good” hash function: entries to be uniformly distributed across N cells
- To store n entries, the expcted number of keys in a bucket would be ceiling(n/N) – O(1) if n is O(N)
- Rehashing -> dynamically growing and shrinking the underlying array
  - __setitem__ and __getitem__: O(1) amortised
- “poor” hash function: all items in the same bucket
  - O(n) performance for separate chaining and open addressing

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__	$O(n)$	$O(n)$	$O(n)$

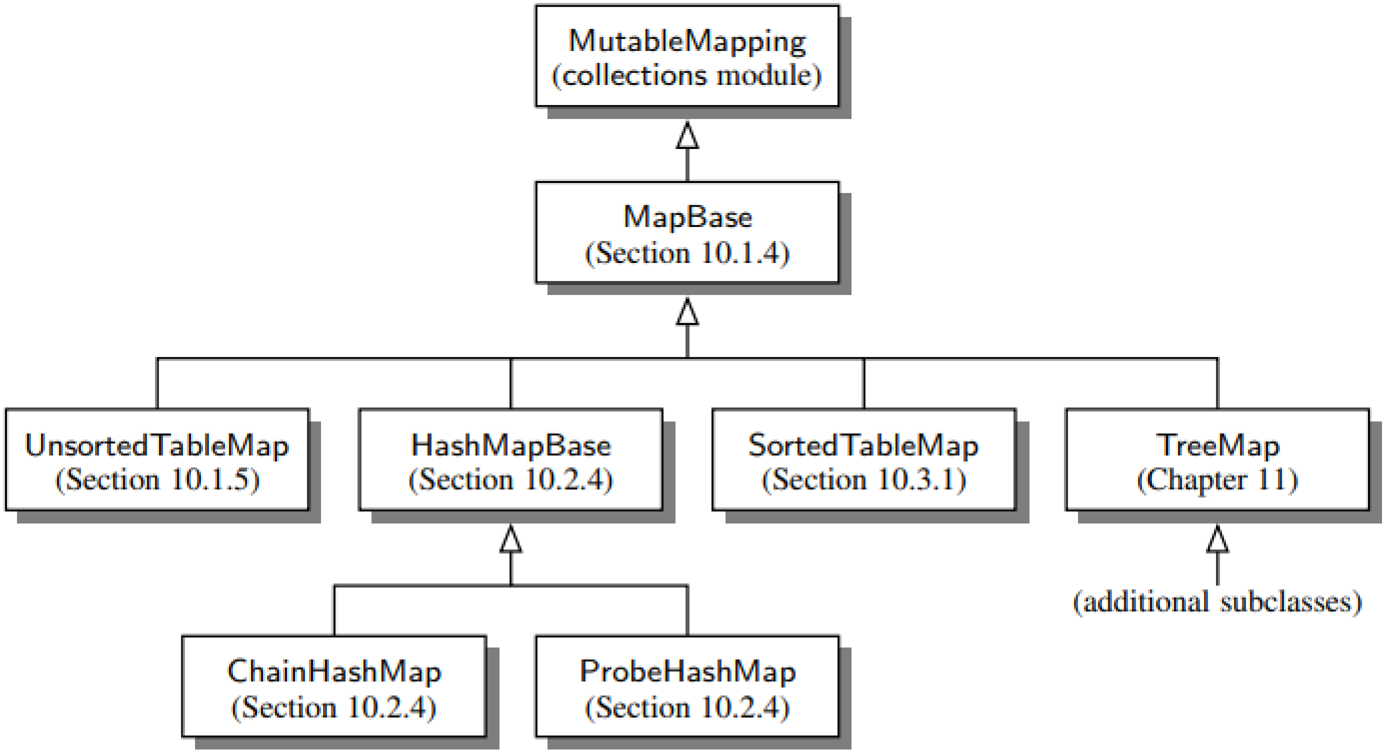
EFFICIENCY OF HASH TABLES

- Python's dict class: implemented with hashing
- Python interpreter
  - Dictionaries to retrieve an object referenced by an identifier in a given namespace
  - c = a + b: __getitem__ for a and b, and __setitem__ for c

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 IMPLELENTATION IN PYTHON

- HashMap base to extend MapBase
- Bucket array represented by a python list
- _n: number of distinct items stored in the hash table
- Table doubling when load factor grows larger than 0.5
- _hash_function: Python's built-in hash function and MAD for compression



HASH TABLE IMPLELENTATION IN PYTHON

- _bucket_getitem(j, k): search bucket j for an item having key k, then returns the associated value, otherwise error
- _bucket_setitem(j, k, v): modify bucket j so that key k is associated with value v. if a new item is inserted, _n is increased
- _bucket_delitem(j, k): removes the item from bucket j with key k. Error if no such item exists
- __iter__: iterator through all keys of the map

HASH TABLE IMPLELENTATION IN PYTHON

- Constructor: init _table, _n, set _prime, _scale, _shift
- Hash function:
  - MAD formular
  - **((ai+b) mod p) mod N**

```
class HashMapBase(MapBase):
  """Abstract base class for map using hash-table with MAD compression."""

  def __init__(self, cap=11, p=109345121):
    """Create an empty hash-table map."""
    self._table = cap * [ None ]
    self._n = 0                              # number of entries in the map
    self._prime = p                          # prime for MAD compression
    self._scale = 1 + randrange(p−1)         # scale from 1 to p-1 for MAD
    self._shift = randrange(p)               # shift from 0 to p-1 for MAD

  def _hash_function(self, k):
    return (hash(k)*self._scale + self._shift) % self._prime % len(self._table)
```

HASH TABLE IMPLELENTATION IN PYTHON

- Len(): returns _n
- __getitem__():get item with key k
- __setitem__(): set/add item, grown array when necessary

```
def __len__(self):
  return self._n

def __getitem__(self, k):
  j = self._hash_function(k)
  return self._bucket_getitem(j, k)          # may raise KeyError

def __setitem__(self, k, v):
  j = self._hash_function(k)
  self._bucket_setitem(j, k, v)              # subroutine maintains self._n
  if self._n > len(self._table) // 2:        # keep load factor <= 0.5
    self._resize(2 * len(self._table) - 1)   # number 2^x - 1 is often prime
```

HASH TABLE IMPLELENTATION IN PYTHON

- _resize():
  - Get old array
  - Create new array
  - Reset _n
  - Insert items

```
def __delitem__(self, k):
  j = self._hash_function(k)
  self._bucket_delitem(j, k)                # may raise KeyError
  self._n -= 1

def _resize(self, c):                       # resize bucket array to capacity c
  old = list(self.items())                  # use iteration to record existing items
  self._table = c * [None]                  # then reset table to desired capacity
  self._n = 0                               # n recomputed during subsequent adds
  for (k,v) in old:
    self[k] = v                             # reinsert old key-value pair
```

SEPARATE CHAINING

- _bucket_getitem:
  - Get bucket, raise error when necessary
  - Return value associated with k
- _bucket_setitem:
  - Create bucket when j does not exist
  - Insert (k, v)
  - Update _n if necessary

```
class ChainHashMap(HashMapBase):
  """Hash map implemented with separate chaining for collision resolution."""

  def _bucket_getitem(self, j, k):
    bucket = self._table[j]
    if bucket is None:
      raise KeyError('Key Error: ' + repr(k))        # no match found
    return bucket[k]                                 # may raise KeyError

  def _bucket_setitem(self, j, k, v):
    if self._table[j] is None:
      self._table[j] = UnsortedTableMap( )         # bucket is new to the table
    oldsize = len(self._table[j])
    self._table[j][k] = v
    if len(self._table[j]) > oldsize:              # key was new to the table
      self._n += 1                                 # increase overall map size
```

SEPARATE CHAINING

- _bucket_delitem:
  - Get bucket, raise error when necessary
  - Delete (k, v) from bucket
- __iter__:
  - Collect items from each bucket

```
def _bucket_delitem(self, j, k):
  bucket = self._table[j]
  if bucket is None:
    raise KeyError('Key Error: ' + repr(k))          # no match found
  del bucket[k]                                        # may raise KeyError

def __iter__(self):
  for bucket in self._table:
    if bucket is not None:                             # a nonempty slot
      for key in bucket:
        yield key
```

LINEAR PROBING

- _AVAIL = object to flag that a bucket is available
- _is_available: if bucket points to None or _AVAIL, returns true. Otherwise false

```
class ProbeHashMap(HashMapBase):
  """Hash map implemented with linear probing for collision resolution."""
  _AVAIL = object( )        # sentinal marks locations of previous deletions

  def _is_available(self, j):
    """Return True if index j is available in table."""
    return self._table[j] is None or self._table[j] is ProbeHashMap._AVAIL
```

LINEAR PROBING

- Probing algorithm
- Check if j is available
- If _table[j] points to None, search has failed (this in theory should not happen)
- Otherwise, keep probing until a match is found

```
def _find_slot(self, j, k):
    """Search for key k in bucket at index j.

    Return (success, index) tuple, described as follows:
    If match was found, success is True and index denotes its location.
    If no match found, success is False and index denotes first available slot.
    """
    firstAvail = None
    while True:
        if self._is_available(j):
            if firstAvail is None:
                firstAvail = j                          # mark this as first avail
            if self._table[j] is None:
                return (False, firstAvail)              # search has failed
        elif k == self._table[j]._key:
            return (True, j)                            # found a match
        j = (j + 1) % len(self._table)                  # keep looking (cyclically)
```

LINEAR PROBING

- _bucket_getitem():
  - Find slot first
  - If no slots found, raise error
  - Return found item
- _bucket_setitem():
  - Find item with linear probing
  - If not found, create an item and update _n
  - Else update value on found item

```
def _bucket_getitem(self, j, k):
  found, s = self._find_slot(j, k)
  if not found:
    raise KeyError('Key Error: ' + repr(k))          # no match found
  return self._table[s]._value

def _bucket_setitem(self, j, k, v):
  found, s = self._find_slot(j, k)
  if not found:
    self._table[s] = self._Item(k,v)                 # insert new item
    self._n += 1                                     # size has increased
  else:
    self._table[s]._value = v                        # overwrite existing
```

LINEAR PROBING

- _bucket_delitem():
  - Find item
  - If not found raise error
  - If found replace item with _AVAIL
- __iter__():
  - Return items except for None and AVAIL

```
def _bucket_delitem(self, j, k):
  found, s = self._find_slot(j, k)
  if not found:
    raise KeyError('Key Error: ' + repr(k))       # no match found
  self._table[s] = ProbeHashMap._AVAIL             # mark as vacated

def __iter__(self):
  for j in range(len(self._table)):                # scan entire table
    if not self._is_available(j):
      yield self._table[j]._key
```

SORTED MAPS

- Map we covered so far – unsorted
  - Exact search: value look-up using a given key
- Sometimes, we want to search for a key from a map
  - E.g. use time stamps as keys to get events that happened in the time stamps
  - For unsorted map: O(N)
- Sorted Map
  - Map with continuous keys in a sorted order
  - We can get efficient operations on Sorted Maps

SORTED MAPS

- Sorted Maps ADT
- M.find_min(): returns the (k, v) pair with the minimum key
- M.find_max(): returns the (k, v) pair with the maximum key
- M.find_lt(k): returns the (k, v) pair, such that its key is less than k
- M.find_le(k): returns the (k, v) pair, such that its key is less than or equal to k
- M.find_gt(k): returns the (k, v) pair, such that its key is greater than k
- M.find_ge(k): returns the (k, v) pair, such that its key is greater than or equal to k
- M.find_range(start, stop): get all (k, v) pairs from start to stop
- Iter(M)
- Reversed(M)

- Sorted Table Map
- Binary search to find the index that stores the (k, v) pair

```
class SortedTableMap(MapBase):
  """Map implementation using a sorted table."""

  #------------------------------ nonpublic behaviors ------------------------------
  def _find_index(self, k, low, high):
    """Return index of the leftmost item with key greater than or equal to k.

    Return high + 1 if no such item qualifies.

    That is, j will be returned such that:
       all items of slice table[low:j] have key < k
       all items of slice table[j:high+1] have key >= k
    """
    if high < low:
      return high + 1                                  # no element qualifies
    else:
      mid = (low + high) // 2
      if k == self._table[mid]._key:
        return mid                                     # found exact match
      elif k < self._table[mid]._key:
        return self._find_index(k, low, mid − 1)       # Note: may return mid
      else:
        return self._find_index(k, mid + 1, high)      # answer is right of mid
```

SORTED MAPS

- Sorted Table Map
- __init__
- __len__
- __getitem__:
  - Use binary search to get the value associated to key k

```
def __init__(self):
  """Create an empty map."""
  self._table = [ ]

def __len__(self):
  """Return number of items in the map."""
  return len(self._table)

def __getitem__(self, k):
  """Return value associated with key k (raise KeyError if not found)."""
  j = self._find_index(k, 0, len(self._table) − 1)
  if j == len(self._table) or self._table[j]._key != k:
    raise KeyError('Key Error: ' + repr(k))
  return self._table[j]._value
```

SORTED MAPS

- __setitem__:
  - Binary search to get the index
  - Update value if k exists
  - Insert (k, v) pair if k does not exist
- __delitem__:
  - Binary search to get the index
  - Pop item if index is legal

```
def __setitem__(self, k, v):
  """Assign value v to key k, overwriting existing value if present."""
  j = self._find_index(k, 0, len(self._table) − 1)
  if j < len(self._table) and self._table[j]._key == k:
    self._table[j]._value = v                          # reassign value
  else:
    self._table.insert(j, self._Item(k,v))             # adds new item

def __delitem__(self, k):
  """Remove item associated with key k (raise KeyError if not found)."""
  j = self._find_index(k, 0, len(self._table) − 1)
  if j == len(self._table) or self._table[j]._key != k:
    raise KeyError('Key Error: ' + repr(k))
  self._table.pop(j)                                   # delete item
```

SORTED MAPS

- find_min:
  - Get (k, v) at index 0
- find_max:
  - Get (k, v) at index n-1

```python
def __reversed__(self):
  """Generate keys of the map ordered from maximum to minimum."""
  for item in reversed(self._table):
    yield item._key

def find_min(self):
  """Return (key,value) pair with minimum key (or None if empty)."""
  if len(self._table) > 0:
    return (self._table[0]._key, self._table[0]._value)
  else:
    return None

def find_max(self):
  """Return (key,value) pair with maximum key (or None if empty)."""
  if len(self._table) > 0:
    return (self._table[−1]._key, self._table[−1]._value)
  else:
    return None
```

SORTED MAPS

- find_ge:
  - Binary search to get index for k
  - Return (k, v) pair if legal
- find_ls:
  - Binary search to get index for k
  - Return (k, v) pair before the index for k if index is legal

```
def find_ge(self, k):
  """Return (key,value) pair with least key greater than or equal to k."""
  j = self._find_index(k, 0, len(self._table) - 1)             # j's key >= k
  if j < len(self._table):
    return (self._table[j]._key, self._table[j]._value)
  else:
    return None

def find_lt(self, k):
  """Return (key,value) pair with greatest key strictly less than k."""
  j = self._find_index(k, 0, len(self._table) - 1)             # j's key >= k
  if j > 0:
    return (self._table[j-1]._key, self._table[j-1]._value) # Note use of j-1
  else:
    return None
```

- find_gt:
  - Binary search to get index for k
  - Increment index j by 1
  - If j is legal return (k, v) at j
- find_range:
  - Binary search to get start
  - Then collecting items from start to stop

```python
def find_gt(self, k):
  """Return (key,value) pair with least key strictly greater than k."""
  j = self._find_index(k, 0, len(self._table) - 1)           # j's key >= k
  if j < len(self._table) and self._table[j]._key == k:
    j += 1                                                  # advanced past match
  if j < len(self._table):
    return (self._table[j]._key, self._table[j]._value)
  else:
    return None

def find_range(self, start, stop):
  """Iterate all (key,value) pairs such that start <= key < stop.

  If start is None, iteration begins with minimum key of map.
  If stop is None, iteration continues through the maximum key of map.
  """
  if start is None:
    j = 0
  else:
    j = self._find_index(start, 0, len(self._table)-1)       # find first result
  while j < len(self._table) and (stop is None or self._table[j]._key < stop):
    yield (self._table[j]._key, self._table[j]._value)
    j += 1
```

SORTED MAPS

- Analysis
- k in M is O(log n) due to binary search
- M[k] = v is O(n) if k does not exists; O(log n) otherwise
- del M[k] is O(n) worst case
- M.find_range(): O(s + log n)

Operation	Running Time
len(M)	$O(1)$
k in M	$O(\log n)$
M[k] = v	$O(n)$ worst case; $O(\log n)$ if existing k
del M[k]	$O(n)$ worst case
M.find_min(), M.find_max()	$O(1)$
M.find_lt(k), M.find_gt(k) M.find_le(k), M.find_ge(k)	$O(\log n)$
M.find_range(start, stop)	$O(s + \log n)$ where s items are reported
iter(M), reversed(M)	$O(n)$

APPLICATIONS OF SORTED MAPS

- Flight databases
- Query on flight database to find flights
  - “from” and “to” cities
  - Departure date
  - Departure time
- Flight database as map, Flight objects as keys
  - k = (from, to, date, time)
- find_ge, find_lt, etc to get a query result
- find_range() to get all feasible flights

(ORD, PVD, 05May, 09:53)	:	(AA 1840, F5, Y15, 02:05, $251),
(ORD, PVD, 05May, 13:29)	:	(AA 600, F2, Y0, 02:16, $713),
(ORD, PVD, 05May, 17:39)	:	(AA 416, F3, Y9, 02:09, $365),
(ORD, PVD, 05May, 19:50)	:	(AA 1828, F9, Y25, 02:13, $186)

QUIZ FOR THIS WEEK

- A 100 bottles
- One of them contains poison, if a rat takes the poison, it dies in 3 days
- The other 99 bottles contain just water
- Question: the minimal number of rats to determine which bottle contains the poison.

THANKS

See you in the next session!