MAPS AND HASH TABLES

School of Artificial Intelligence

PREVIOUSLY ON DS&A

- Priority Queues
- Heaps

TREES

- Proper binary tree （真二叉树）：二叉树的每一个结点都有0个或2个子节点
- Full binary tree（满二叉树）：除叶子结点外，所有内部结点都有2个子结点；所有叶子结点的深度都相同
- Complete binary tree（完整二叉树）：对于高为h的二叉树，其从0层到h-1都有2^i个结点（i为层数）；对于第h层，如果没有2^h个结点的话，则所有结点都集中在第h层的最左面

PRIORITY QUEUES （优先队列）

- Collection of prioritized elements
  - Arbitrary element insertion
  - Removal of the element that has first priority
  - When an element is added, a priority can be assigned to it with a **key**
  - Element with the minimum key will be removed next from the queue
  - **Key**s can be other data types, as long as there is a way to compare them
  - E.g. a < b for instances a and b
- Implementation of Priority Queues
  - Based on Positional list
    - Can you do with an array? What problems can arise?
  - Unsorted list
    - Add – O(1), remove_min - O(n)
  - Sorted list
    - Add – O(n), remove_min – O(1)

PRIORITY QUEUES （优先队列）

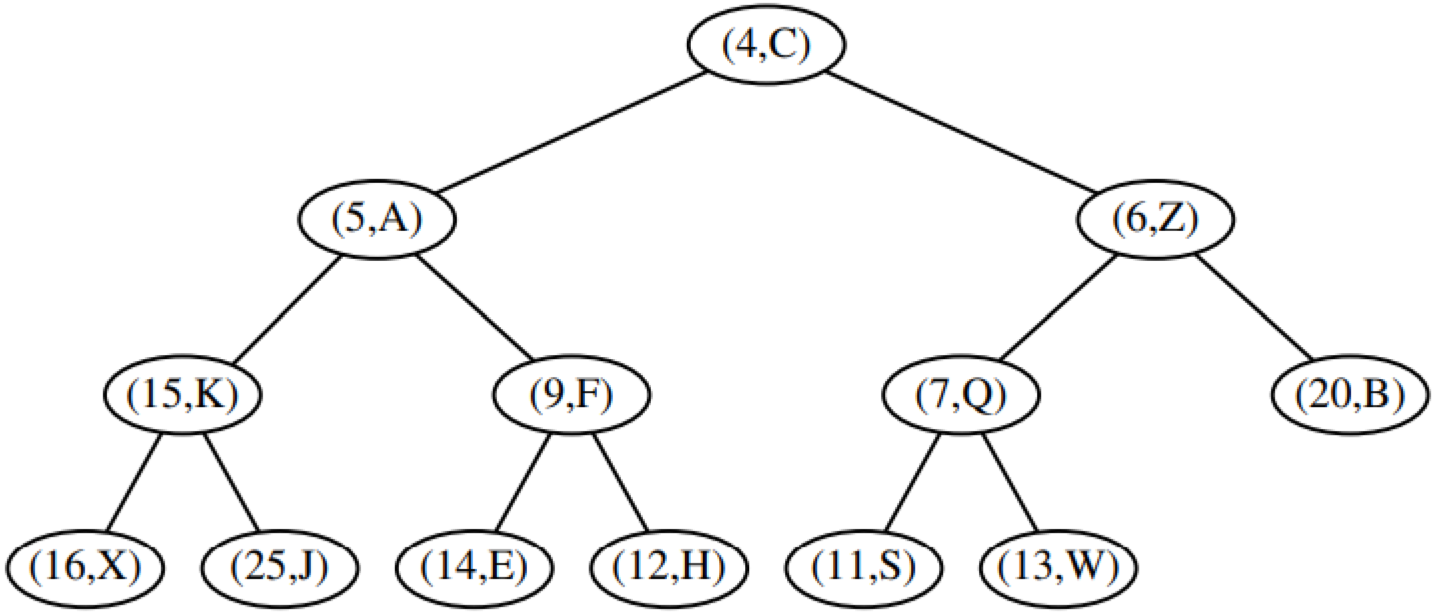
- Abstract Data Type (ADT)
- P.add(k, v): insert an item with key k and value v into priority queue P
- P.min(): returns a tuple (k, v), representing the key and value of an item in the priority queue P with the minimal key (but do not remove the item)
  - Error when the queue is empty
- P.remove_min(): remove an item with the minimal key, return a tuple (k, v), representing the key-value pair to be removed
  - Error when the queue is empty
- P.is_empty(): returns true when P has no items
- len(p): returns the number of items in the priority queue P

HEAP（堆）

- Heap: a binary tree that stores a collection of items at its positions
  - A relational property defined in terms of the way keys are stored in T
  - A structural property defined in terms of the shape of T itself
- Relational property (**heap order property**): In a heap T, for every position p other than the root, the key stored at p is greater than or equal to the key stored at p's parent
- Structural property (**complete binary tree property**): A heap T with height h is a complete binary tree if levels 0, 1, 2, …, h-1 of T have the maximum number of nodes possible (level i has 2^i nodes, for 0<= i <= h-1) and the remaining nodes at level h reside in the lfeftmost possible positions at that level

HEAP（堆）

- Complete
  - Levels 0, 1, and 2 are full
  - 6 nodes in level 3 are in the six leftmost possible positions at that level
- An alternative definition
  - If we are to store a complete binary tree T with n elements in an array A , then its 13 entries would be stored from A[0] to A[n-1]

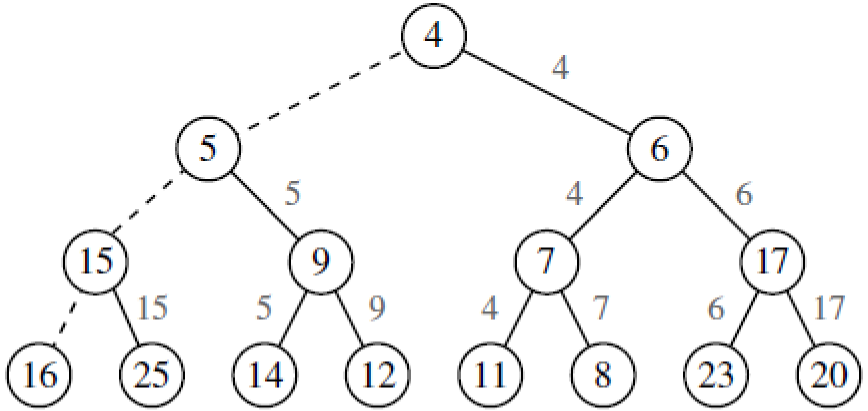


BOTTOM UP HEAP CONSTRUCTION

- Add 16, 15, 4, 12, 6, 7, 23, 20, 25, 9, 11, 17, 5, 8, 14 (15 elements) in order to construct a heap
- h = floor(log n) = 3
- construct (n+1)/2 elementary heap storing one entry each
- In the generic i^{th} step, 2<=i<=h, form $(n+1)/2^i$ heaps, each storing 2^i -1entries, perform downheap() to restore the heap order property

ASYMPTOTIC ANALYSIS OF BOTTOM UP HEAP CONSTRUCTION

- **Bottom up construction of a heap with n entries takes O(n) time, assuming two keys can be compared in O(1) time**
- Primary cost: downheap() at each nonleaf position.
- Let P_v denote the path of T from nonleaf node v to its inorder successor leaf, P_v is proportional to the height of the subtree roted at v.
- Total running time therefore the sum of the sizes of paths
- Paths are edge-disjoint, and therefore bounded by the number of total edges, hence O(n)

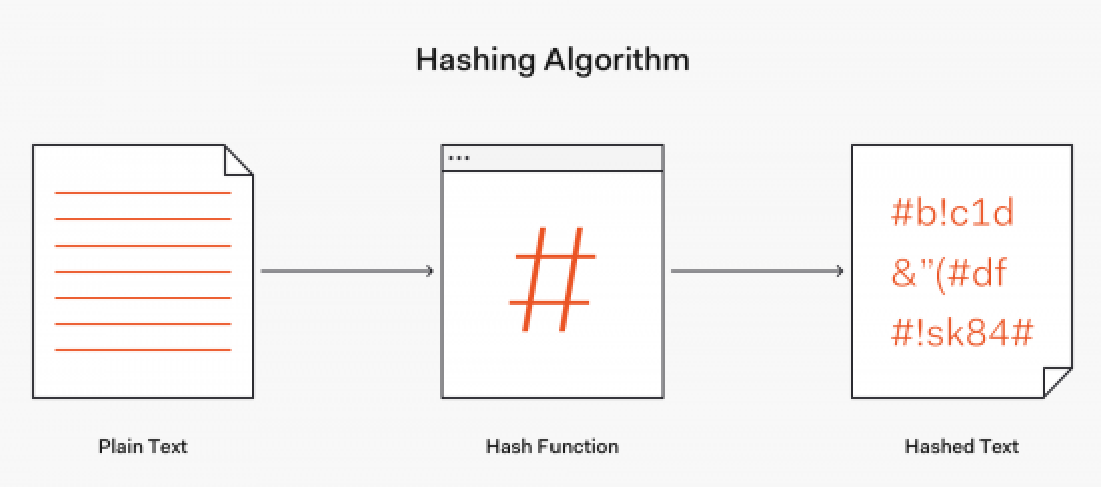


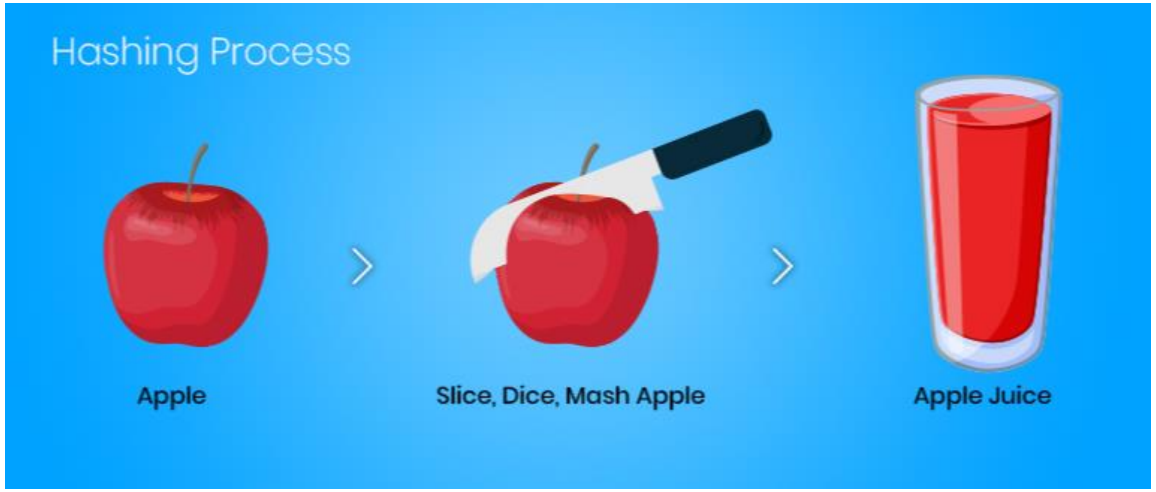
ADAPTABLE PRIORITY QUEUES

- Priority queue ADT is sufficient for most basic applications. However, the following situations are not accounted for:
- A person waiting in queue may want to drop out, requesting to be removed from the waiting list.
  - Need a remove() operation
- An element may suddenly have a higher priority and needs to be placed in its rightful place.
  - Need a update() operation
- The above behaviours make the priority queue adaptable to any situations we can think of

THIS LECTURE

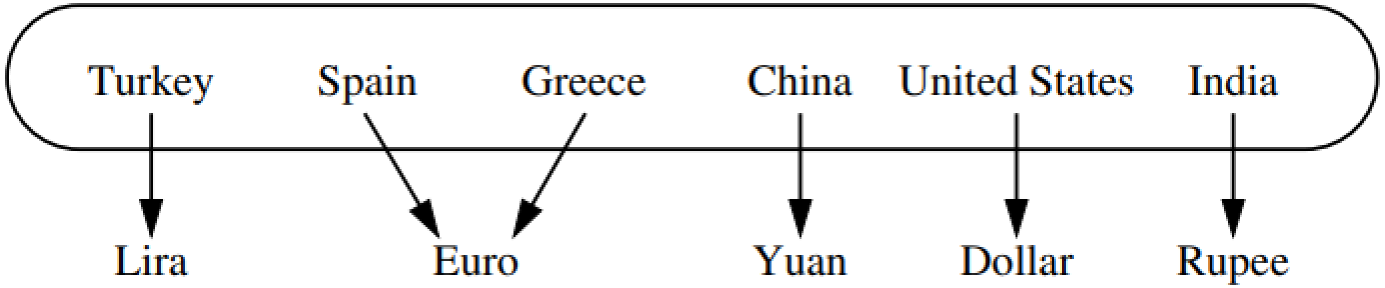
- Maps/dictionaries
- Hash Tables
- Hash Functions





MAPS AND DICTIONARIES

- Key->Value pairing
  - Unique association
- Most significant data structure in any programming language
- Often known as **associative arrays** or **maps**
- Keys are (assumed) to be unique, values are not necessarily unique
- Python: dict class
- Use key as ‘index’
- Indices need not be consecutive nor numeric



MAPS AND DICTIONARIES

- Common applications
- A university's information system
  - Student ID -> student record (name, address, course grades, etc)
- Domain Name System (DNS)
  - Host name -> IP address
  - abc.com -> 192.144.121.130
- Social media site
  - Username -> user home page
- Computer graphics
  - Colour name -> RGB values
  - "red" -> (255,0,0)
- Programming language name space
  - Pi -> 3.14159

MAPS AND DICTIONARIES

- The Map ADT
- **M[k]**: returns the value v associated with key k in map M, if one exists
  - Raise error if key does not exist
- **M[k] = v**: assign value v with key k in map M, replacing existing value if the map already contains an item with key equal to k
- **del M[k]**: remove from map M the item with key equal to k
  - Raise error if M does not map k
- **len(M)**: number of itmes in map M
- **Iter(M)**: iterator for a map

MAPS AND DICTIONARIES

- The Map ADT
- **k in M**: returns true if map contains an item with key k
- **M.get(k, d=None)**: return M[k] if k exists, otherwise return value d
- **M.setdefault(k, d)**: if key k exists in the map, return M[k], if k does not exist, set M[k] = d and return that value
- **M.pop(k, d=None)**: remove the item associated with key k from the map and return its associated value v. If key is not in the map, return default value d
- **M.popitem()**: remove an arbitrary key-value pair from the map, and return a (k,v) tuple representing the removed pair
  - Error when map is empty

MAPS AND DICTIONARIES

- The Map ADT
- **M.clear():** remove all key-value pairs from the map
- **M.keys()**: return a set of all keys of M
- **M.values()**: return a set view of all values of M
- **M.items()**: return a set of (key, value) tuples for all entries
- **M.update(M2)**: assign M[k] = v for every (k, v) pair in map M2
- **M == M2**: returns true if maps M and M2 have identical items
- **M != M2**: returns true if M and M2 do not have identical items

MAPS AND DICTIONARIES

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}

MAPS AND DICTIONARIES

- Counting word frequencies in a document
- Categorizing an email or news
- Convert document into lower case then split()
- Reconstruct words
- Add words into the dictionary
- Go through the dictionary and perform statistical analysis

```
freq = { }
for piece in open(filename).read().lower().split():
  # only consider alphabetic characters within this piece
  word = ''.join(c for c in piece if c.isalpha())
  if word:     # require at least one alphabetic character
    freq[word] = 1 + freq.get(word, 0)

max_word = ''
max_count = 0
for (w,c) in freq.items():        # (key, value) tuples represent (word, count)
  if c > max_count:
    max_word = w
    max_count = c
print('The most frequent word is', max_word)
print('Its number of occurrences is', max_count)
```

MUTABLEMAPPING ABSTRACT BASE CLASS

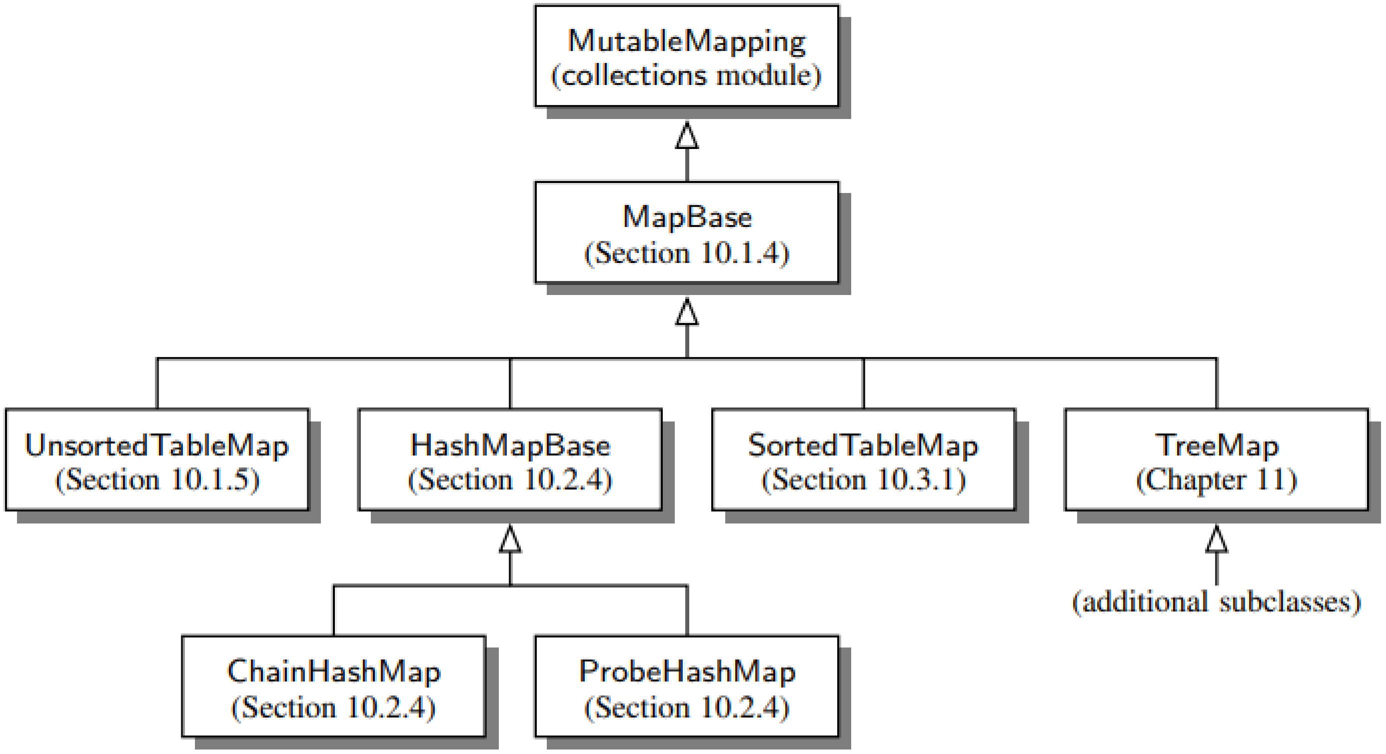
- What's an abstract base class?
- Methods declared to be abstract in a base class must be implemented by its concrete subclasses
- Python's collection module
- Mapping: all nonmutating methods supported by Python's dict class
- MutableMapping: extends Mapping to include mutating methods
- __getitem__, __setitem__, __delitem__, __len__ and __iter__

```
def __contains__(self, k):
  try:
    self[k]
    return True
  except KeyError:
    return False
```

```
def setdefault(self, k, d):
  try:
    return self[k]
  except KeyError:
    self[k] = d
    return d
```

THE MAPBASE CLASS

- MutableMapping class
  - From Python's collections module
- MapBase
  - Extends Mutable Mapping
  - Greater reusability
  - Compositional pattern to group key-value pair in a single instance



THE MAPBASE CLASS

- Internal _Item class
  - To store _key and _value
  - Overrids ==, != and < operator

```
class MapBase(MutableMapping):
  """Our own abstract base class tha

  #------------------------------ nested _
  class _Item:
    """Lightweight composite to stor
    __slots__ = '_key', '_value'

    def __init__(self, k, v):
      self._key = k
      self._value = v

    def __eq__(self, other):
      return self._key == other._key

    def __ne__(self, other):
      return not (self == other)

    def __lt__(self, other):
      return self._key < other._key
```

UNSORTED MAP IMPLEMENTATION

- UnsortedTableMap
- Subclass of MapBase to store key-value pairs in unsorted order in a Python list
- __getitem__: M[k]
- __setitem__: M[k] = v

```
class UnsortedTableMap(MapBase):
  """Map implementation using an unordered list."""

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

  def __getitem__(self, k):
    """Return value associated with key k (raise KeyE
    for item in self._table:
      if k == item._key:
        return item._value
    raise KeyError('Key Error: ' + repr(k))

  def __setitem__(self, k, v):
    """Assign value v to key k, overwriting existing val
    for item in self._table:
      if k == item._key:
        item._value = v
        return
    # did not find match for key
    self._table.append(self._Item(k,v))
```

THE MAPBASE CLASS

- __getitem__: M[k]
- __setitem__: M[k] = v
- __delitem__: del M[k]
- __len__: len(M)

- Complexity: O(n)

```
def __delitem__(self, k):
  """Remove item associated with key k (raise KeyError if not
  for j in range(len(self._table)):
    if k == self._table[j]._key:                              # Fo
      self._table.pop(j)                                      # rer
      return                                                  # an
  raise KeyError('Key Error: ' + repr(k))

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

def __iter__(self):
  """Generate iteration of the map's keys."""
  for item in self._table:
    yield item._key                                           # yie
```

HASH TABLES (哈希表)

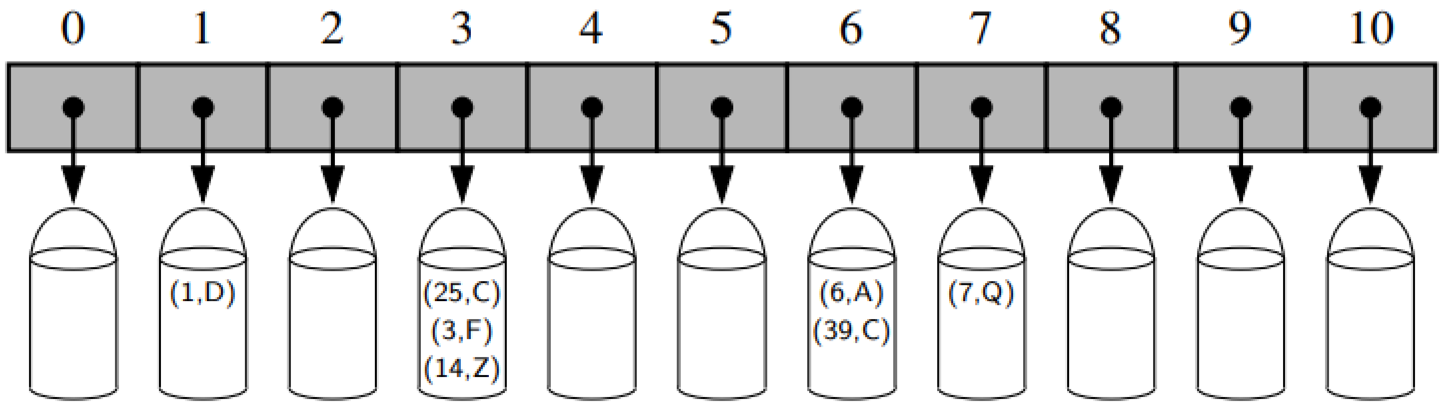
- Most practical data structures for implementing a map
- A map M supports the abstraction of using keys as indices with a syntax such as M[k].
- Assume a map with n items uses integer keys from 0 to N-1 for some N >= n
- We can represent the map like this:

0	1	2	3	4	5	6	7	8	9	10
	D		Z			C	Q			

- __getitem__, __setitem__ and __delitem__ become O(1)
- Problems?

HASH TABLES (哈希表)

- Problems
  - Keys n may not be continuous, therefore an array for the map may have size N >> n
  - A map's key can be other data types, not just integers
- Solution: hash function to map keys to corresponding indices in a table
- Ideally, keys will be distributed in the range from 0 to N-1
- But, there may be two or more distinct keys that get mapped to the same index
- Bucket array

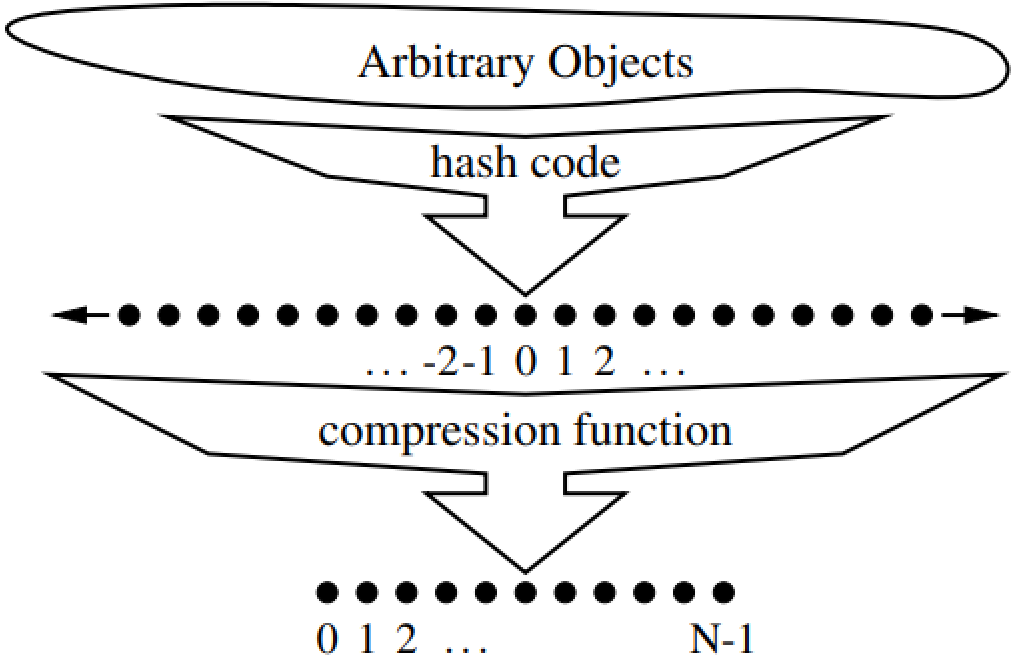


HASH FUNCTION（哈希函数）

- Hash function - **h**: to map each key **k** to an integer in the range [0, **N**-1], where **N** is the capacity of the bucket array for a hash table.
- h(k) produces a value, which can be used as an index into the array, A, instead of the key k.
- (k, v) in the bucket array would be A[h(k)]
- When two or more keys have the same hash value, they will be mapped to the same bucket in A.
  - A **hash collision**
- A hash function is “good” if it maps the keys with sufficiently few collisions
- Hash function also needs to be fast and easy

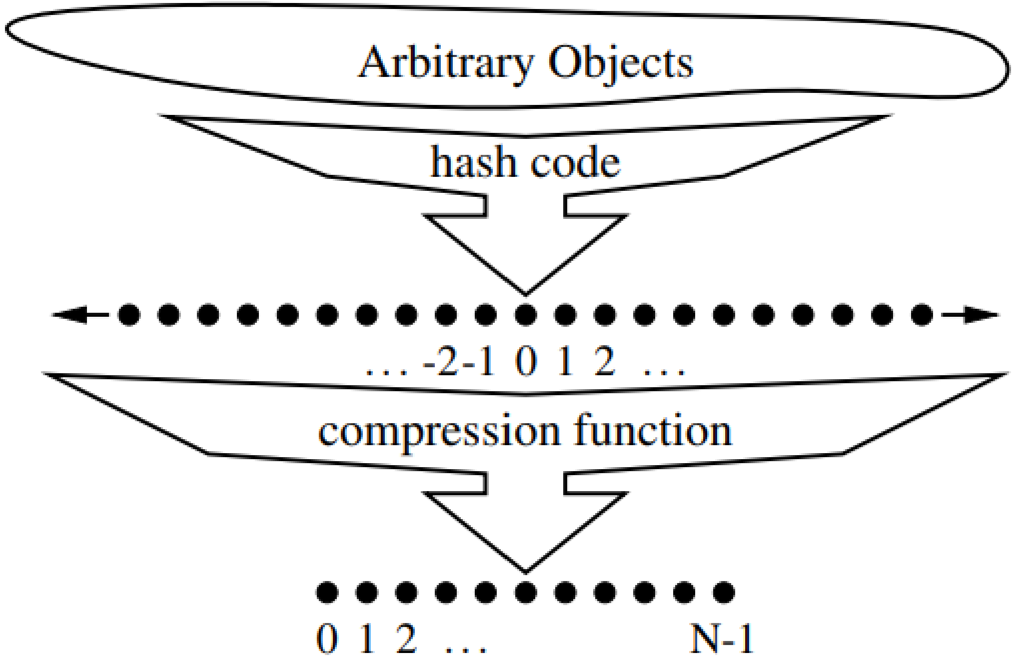
HASH FUNCTION（哈希函数）

- Hash function: h(k)
- **Hashing**: produce a *hash code* that maps a key k to an integer
- **Compressing**: maps the hash code to an integer within a range of indices [0, N-1]
- Why the separation?
  - Independence: hashing is independent of a specific hash table size
  - OO design: hash functions can be overridden



HASH CODES（哈希码）

- Hashing: take an key k and compute an integer that is called the **hash code** for k
- k does not need to be in the range [0, N-1]
- Hash codes produced should avoid collisions as much as possible



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

POLYNOMIAL HASH CODES （多项式哈希码）

- Polynomial – to spread out the influence of each component across the resulting hash code
- For constant a, its polynomial value will periodically overflow the bits used for an integer, but is often ignored
- Therefore a should have nonzero, low-order bits.
- 33, 37, 39, 41 are good choices for a when working with character strings (English)
  - 50,000 English words formed as the union of the word lists for different version of Unix, using a = 33, 37, 39 or 41 produces less than 7 collisions

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:
- <u>00111</u>101100101101010100010101000 to 101100101101010100010101000<u>00111</u>
- 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

HASH CODES IN PYTHON

- hash(x): returns an integer hash code for object x
- Only <u>immutable</u> data types are hashable
- Why?
  - The hash code of an object remains constant
- Important: the hash code of an object should remain the same during the object's life time
- Immutable data types: int, float, str, tuple, frozenset
- For character strings: similar to polynomial hash codes, but uses exclusive-or computations than additions

HASH CODES IN PYTHON

- For mutable objects, especially the objects that are instances of your classes
- Custom the hash() function:

```
def __hash__(self):
  return hash( (self._red, self._green, self._blue) )    # hash combined tuple
```

- But at the same time, you should also define: == operator
  - If x == y, then hash(x) == hash(y)
- If 5 == 5.0, then hash(5) == hash(5.0)

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

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!