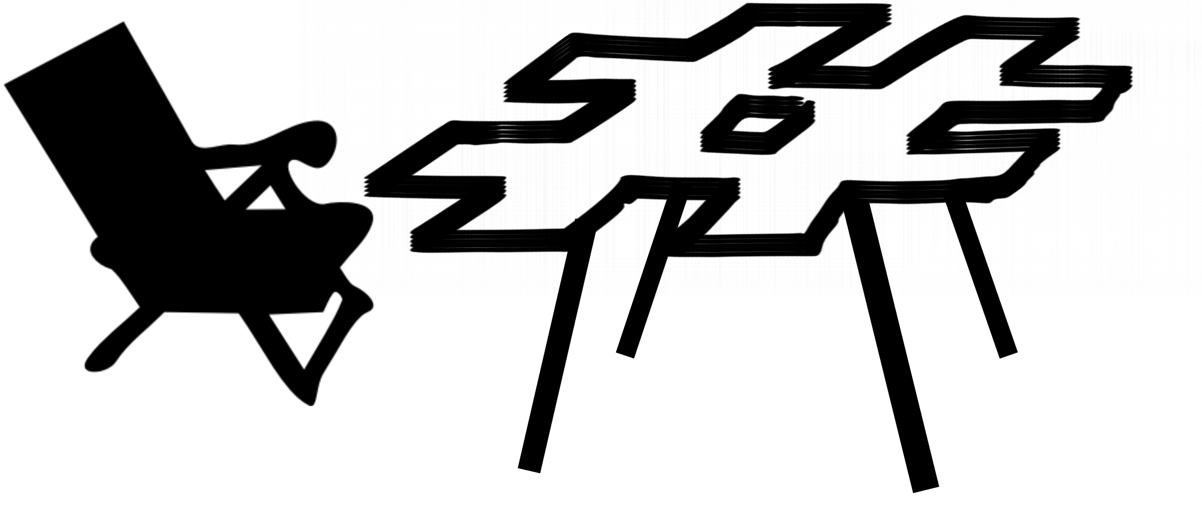
CS 240
Fall 2014

Mike Lam, Professor



Hash Tables

Hash Tables

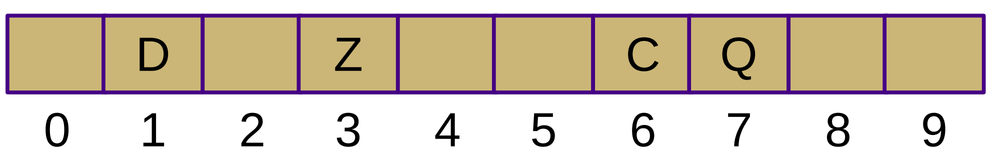
- Data structure for fast key/value lookups
  - Used to implement the Map ADT
  - Goal: *O(1)* access (insert/modify/delete)
- Observation: arrays provide *O(1)* access
  - How to map from keys to array indices?
  - How large does the array need to be?

Map ADT **Array**

Hash Tables

- Simple case: keys are integers in [0, *N*)
  - Create an array of length *N*
  - Use keys directly as indices into the array
- This does not scale!
  - *N* could be very large
  - Keys might not be integers

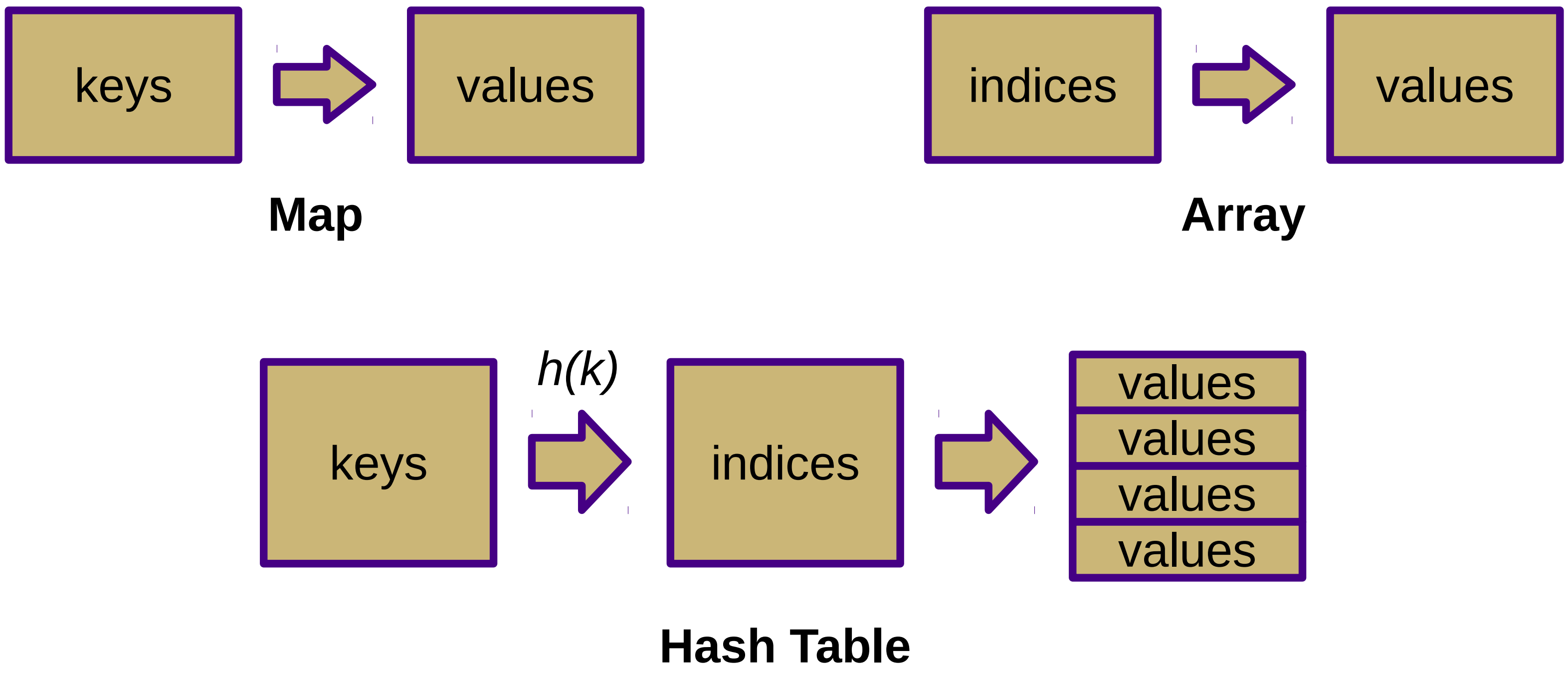
```
ht = Array()
ht[3] = "Z"
ht[6] = "C"
ht[7] = "Q"
ht[1] = "D"
```



Items: (1,D), (3,Z), (6,C), (7,Q)

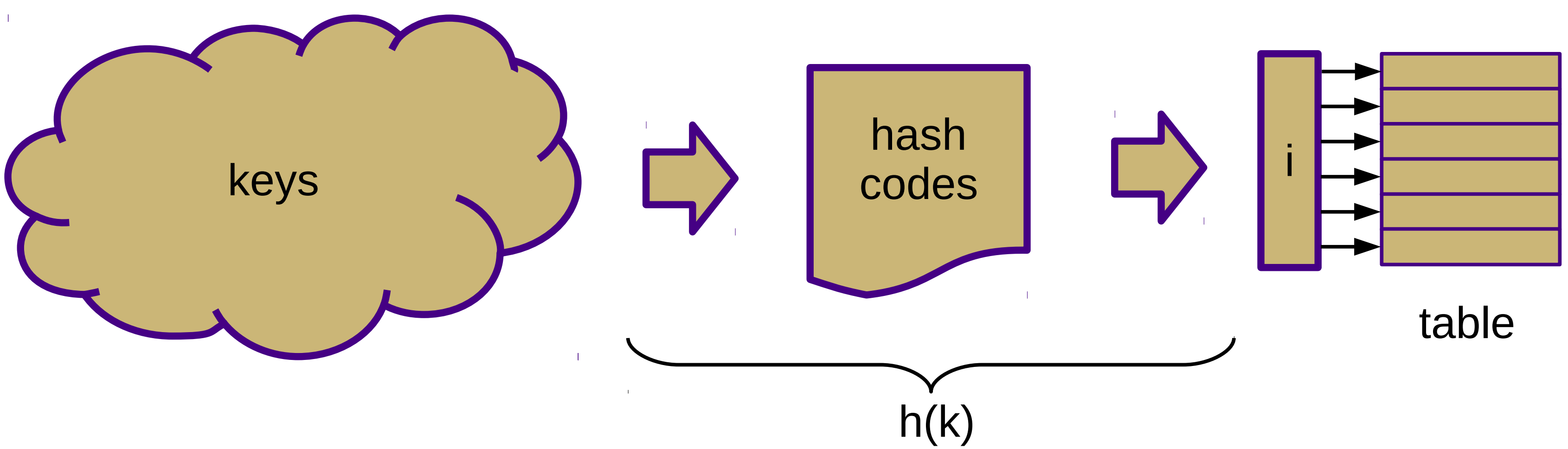
Hash Tables

- Main concept: hash function
  - Maps keys → table indices
  - Table holds "buckets" of elements



Hash Functions

- Hash code (key → 32/64-bit integer)
  - Translation from key domain to hash code domain
  - Key can be any immutable object
  - Hash codes are usually native integers
- Compression function (hash code → table index)
  - Compression from hash code domain to index domain
  - Result is used to access table storage



Hash Functions

- Major problem: Collisions
  - Multiple keys mapping to the same index
  - Two-fold approach:
    - Minimize collisions by choosing a good hash code
    - Handle collisions with chaining or probing

Hash Codes

- Most codes are based on interpreting raw bits as integers
  - Issue: key size may be greater than native integer width
- Need to combine multiple integers
  - Truncation
  - Summation
  - Exclusive-or (XOR)

  (Truncation, Summation, Exclusive-or: bad for variable-length objects)
  - Polynomial combination
  - Cyclic shifting
  - Cryptographic hashes (e.g., MD5, SHA-1)

Bitwise Arithmetic

- Integer → bit string representation: `bin(i)`
- Bit string representation → integer: `int(s, 2)`
- Bitwise operations
  - AND: `x & y`
  - OR: `x | y`
  - NOT: `~x`
  - XOR: `x ^ y`
  - Left shift: `x << i`
  - Right shift: `x >> i`

Implementation Note

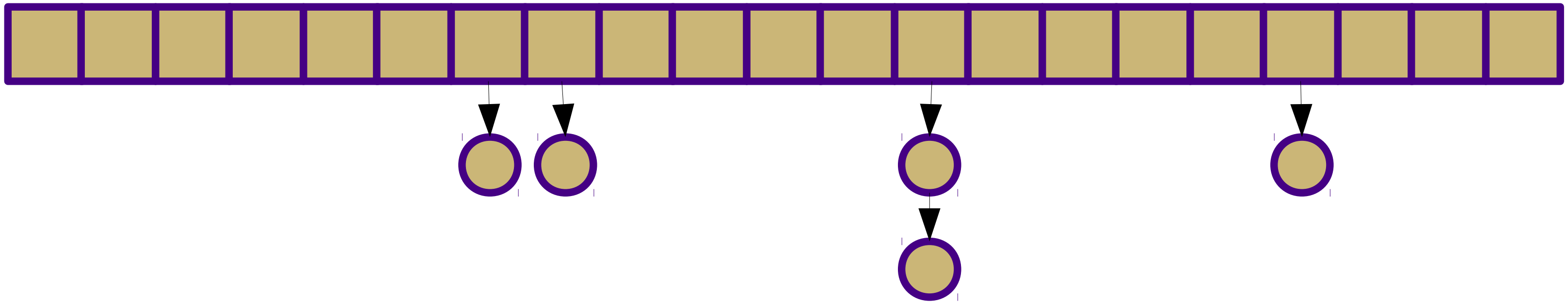
- Dictionary keys in Python must be immutable
  - A key's hash should not change while it is in a dictionary
  - Thus, mutable objects are not good keys
  - In fact, only immutable objects are hashable in Python
    - This is a policy decision
  - Thus, only immutable objects can be used as keys

Compression Functions

- Simplest: Modulus division
  - $h(k) \% N$
  - N is the number of buckets in the hash table
  - N should be a prime number
- Better: Multiply-Add-and-Divide
  - $((a \cdot h(k)) + b) \% p) \% N$
  - p is a prime number larger than N
  - a and b are random integers from $[0, p\text{-}1]$
    - $a > 0$
  - Essentially a pseudo-random number generator that uses hash codes as seeds

Collision Handling

- Separate chaining
  - Each bucket is a linked list of elements
  - Load factor: $\lambda = n/N$
    - Expected size of each bucket
    - If the hash function is good, map operations run in $O(\lambda)$
    - This should be a small constant
      - Preferably less than 1
    - As long as λ is $O(1)$, map operations run in $O(1)$ expected time

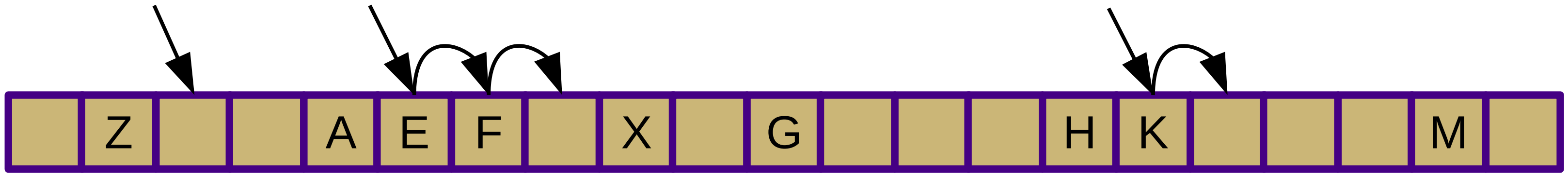


Collision Handling

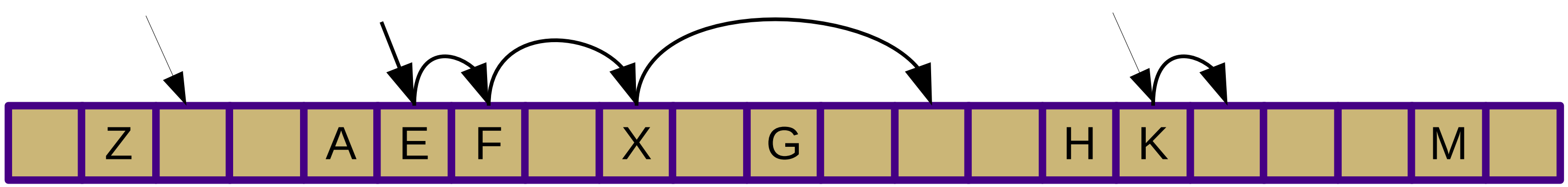
- Open addressing
  - Only one (key, value) pair per "bucket"
  - Problem: $h(k)$ not guaranteed to be open
  - Probing scheme to find an open bucket
  - Load factor: $\lambda = n/N$
    - Percentage of buckets that are occupied
- Approaches
  - Linear probing: $(h(k) + i) \% N$
  - Quadratic probing: $(h(k) + i^2) \% N$
  - Double hashing: $(h(k) + i \cdot h'(k)) \% N$
  - Pseudo-random probing: $(h(k) + prand(i)) \% N$

Open Addressing

- Linear probing



- Quadratic probing



Collision Handling

- Coalesced hashing (hybrid chained/open)
  - Maintain chains as pointers between buckets
  - Avoids some of the overhead of probing
- Cuckoo hashing (multiple hash functions)
  - Use multiple hash functions (primary and alternate)
  - If new key's bucket is full, remove existing key and re-insert it using alternate hash
  - Repeat until empty bucket is found or an infinite loop is detected

Load Factors

- Separate chaining
  - Want to keep λ less than 1 (preferably < 0.9)
- Open addressing
  - Want to keep λ less than 1/2 or 2/3
- Rehashing
  - When constraints above are violated, resize the hash table and re-apply the compression function to re-insert all keys
  - Cost can be amortized by doubling the table size
    - Just as with dynamic arrays

Hashing Analysis

- The expected # of keys in a bucket is *ceil*(*n/N*)
  - This is *O(1)* if n is *O(N)*
  - Assumes a good hash function
  - Assumes enforcement of appropriate load factor
- Thus, expected costs for major map operations (insertion, modification, lookup, removal) are all *O(1)*
  - Worse case: *O(n)*
- Full probabilistic analysis is beyond the scope of this class

Retrospective

- Next PA: implement the Set ADT with a hash table
  - (just kidding!)
- Set/Map equivalence: a Set is a Map with no values
- Progression of Set/Map implementations:
  - Array / Linked list
    - Mostly *O(n)* operations
  - Skip list / Balanced binary tree
    - Mostly *O(log n)* operations
  - Hash table
    - Mostly *O(1)* operations