Lec8_Hasing

Dictionary Problem

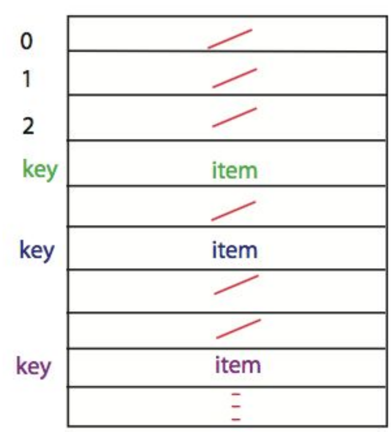
- Abstract Data Type(ADT) - maintain a set of items, each with a key, subject to
  - insert(item) : add item to set
  - delete(item) : remove item from set
  - search(key) : return item with key if it exists
- Goal : O(1) time per operation

Python Dictionaries

- D[key] → search
- D[key] = val → insert
- del D[key] → delete
- Item = (key, value)

Simple Approach : Direct Access Table

→ This means items would need to be stored in an array, indexed by key



- Problems
  - keys must be nonnegative integers (or using two array, integers)
  - large key range ⇒ large space
- Solution to 1 : **“Prehash”** keys to integers
  - In theory, possible because keys are finite ⇒ set of keys is countable

- In Python : hash(object) *(actually hash is misnomer should be "prehash")* where object is a number, string, tuple etc. or object implementing __hash__ (default = id = memory address)
- In theory, x == y ⇔ hash(x) == hash(y)
- Python applies some heuristics for practicality: for example, hash('\0B ') = 64 = hash('\0\0C ')
- Object's key should not change while in table (else cannot find it anymore)
- No mutable objects like lists

- Solution to 2 : **Hashing**
    - Reduce universe U of all keys down to reasonable size m for table
    - <u>Idea</u> : m ~= n (# keys stored in dictionary)
    - <u>hash function</u> h : U → {0,1,2, … , m-1}

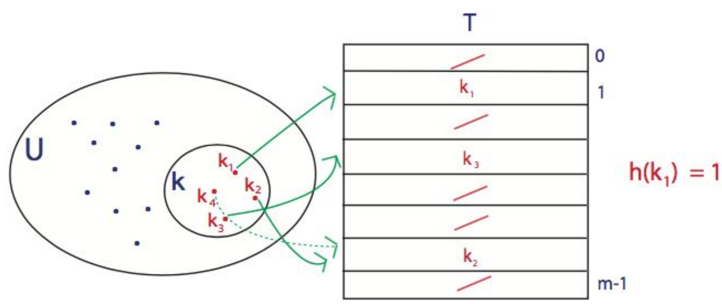


Figure 2: Mapping keys to a table

    - two keys ki, kj ∈ K collide if h(ki) = h(kj)
        - How do we deal with collision? → chaining / open addressing

Chaining

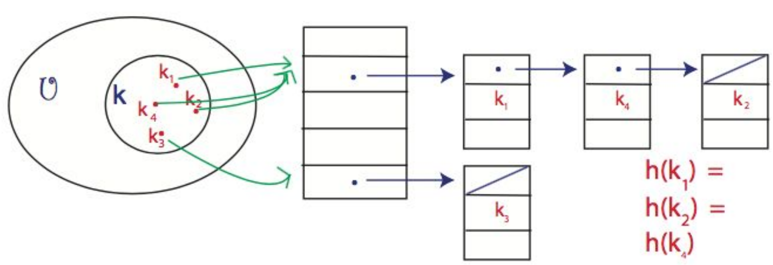


Figure 3: Chaining in a Hash Table

- Search must go through whole list T[h(key)]
- Worst case : all n keys hash to same slot ⇒ Θ(n) per operation

Simple Uniform Hashing

→ An assumption : Each key is equally likely to be hashed to any slot of table, independent of where other keys are hashed.

- let n = # keys are hashed
  m = # slots in table
- load factor α = n/m
  = expected # keys per slot
  = expected length of a chain
- expected running time for search = $\Theta(1+\alpha)$
  1 : apply hash function & random access to slot
  α : search the list
- O(1) if α = O(1) i.e.) m = Ω(n)

Hash Functions

- Division method : h(k) = k mod m
  - Practical when m i preme but not close to power of 2 or 10
- Multiplication Method : h(k) = [(a*k) mod 2^w] >> (w-r)
  a : random
  k : w bits
  m = 2^r
  - Practical when a is odd & 2^(w-1) < a < 2^w & not close

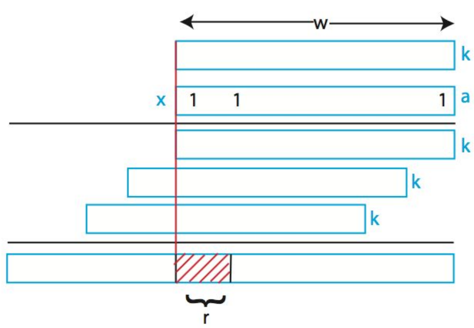


Figure 4: Multiplication Method

⇒ Hatched part is the key

- Universal Hashing : h(k) = [(ak+b) mod p] mod m
  where a and b are random ∈ { 0,1,2 ….p-1 } and p is a large prime (> |U|).
  This implies that for worst case keys k1 /= k2, (*and for a,b choice of h*) :
  Pra,b{ event X k1k2 } = Pra,b { h(k1) = h(k2)} = 1/m