Data Structures

Hashtables

Hashing

A completely different approach to searching from the comparison-based methods (binary search, binary search trees)

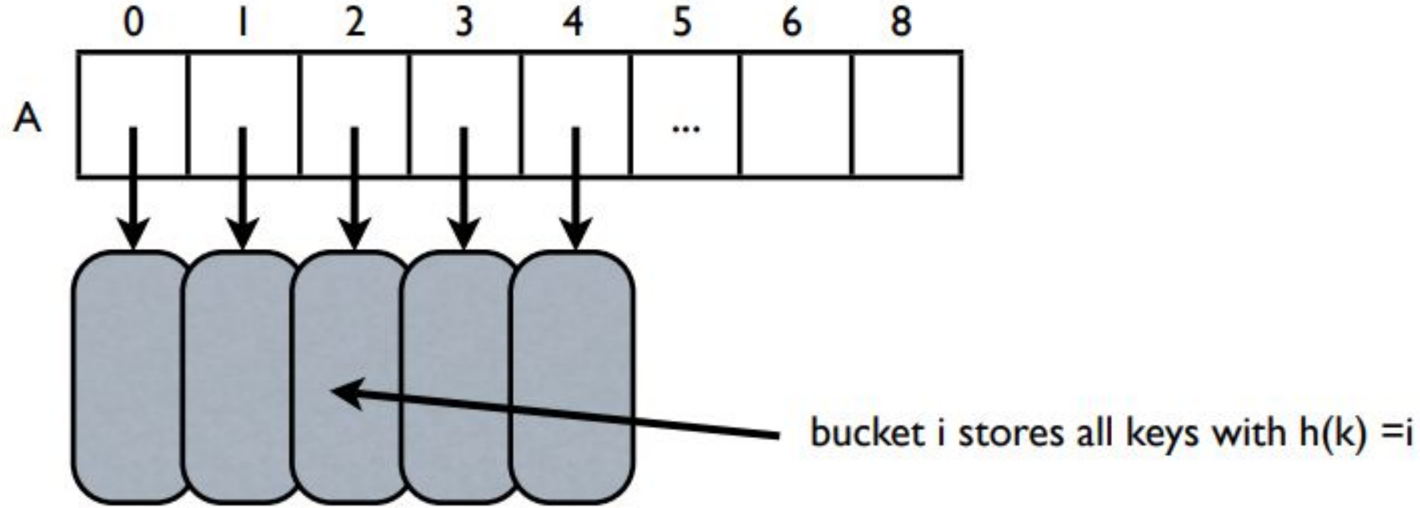
- rather than navigating through a dictionary data structure comparing the search key with the elements, hashing tries to reference an element in a table directly based on its key
- hashing transforms a key into a table address

Hashing

Hashing has two components:

- The hash table: an array A of size N
  - Each entry is thought of a bucket: a bucket array
- A hash function maps each key to a bucket
  - h is a function: {all possible keys} -> {0, 1, 2, .., N - 1}
  - key k is stored in bucket h(k)
- The size of the table N and the hash function are decided by the user

Hashing



Example

- Keys: integers
- Choose N = 10
- Choose h(k) = k % 10
  - [k % 10 is the remainder of k / 10]
- Add (2,*), (13,*), (15,*), (88,*), (2345,*), (100,*)
- Collision: two keys that hash to the same value
  - E.g. 15, 2345 hash to slot 5

Hashing

- h: {universe of all possible keys} —> {0, 1, 2, .., N-1}
- The keys need not be integers:
  - E.g. strings
  - Define a hash function that maps strings to integers
- Hashing is an example of space-time tradeoff:
  - If there were no memory (space) limitation, simply store a huge table
    - O(1) search, insert, delete
  - If there were no time limitation, use linked list and search sequentially
- Hashing: use a reasonable amount of memory and strike a balance space-time
  - Adjust hash table size
  - Under some assumptions, hashing supports insert, delete and search in in O(1) time

Hashing

- Notation
  - U = universe of keys
  - N = hash table size
  - n = number of entries
    - Note: n may be unknown beforehand
- Goal of a hash function:
  - The probability of any two keys hashing to the same slot is 1/N (called “universal hashing”)
  - Essentially this means that the hash function throws the keys uniformly at random into the table