Helping Hand Notes on Hashing

Lee S. Koh

Looking back (what's behind us) and where hashing fits in

- Storage and access techniques (for enhancing problem solving capability)
  - Data structures – storage (for data)
  - (and Algorithms) – access (of data)
- Two categories of storage and access techniques seen so far:
  - Storing data items *sequentially* (data items arranged in the order in which they arrive).
    - Applicable data structures – contiguous (arrays) or non-contiguous (linked lists)
    - Applicable algorithm – linear/sequential/serial search – *O(n)*
  - Storing data items in *sorted* order
    - Applicable data structures – contiguous (arrays) or non-contiguous (BST/AVL)
    - Applicable algorithm – binary search – *O(log n)* with array, shoot for *O(log n)* with BST, *O(log n)* with AVL
- Hashing – a third category of storage and access technique – shoot for *O(1)*
  - Useful technique in numerous (not all) real world situations – storing/accessing data based on ID (*key*).
    - Best for *plain (exact match based on key) search*/*insert*/*retrieve*/*update*/*remove* involving large amount of data.
    - Not good for (to-be-done-based-on-key) *min-max* searches, *range* searches, *rough match* searches and *ordered* visits.
  - ADT that probably was driving force for hashing development -> *table* (aka *map* or *dictionary*).

Perfect hashing example

- `EmpInfo`: `EmpId` from 0 to 99, array of size 100.

More realistic hashing example

- `EmpInfo`: `EmpId` from 0 to 99999, array of size 100.

Lessons/observations/... from hashing examples just seen:

- *Hash function*, *hash value* (*hash code*), *hash table* and *hashing*.
  - What hash function for realistic hashing example.
    - Division type hash function – perhaps most commonly used (for numeric keys) – **`h(key) = key % TABLE_SIZE`**
  - What hash function for perfect hashing example.
- (Compare "store sequentially", "store sorted" and "store hashed": **50003**, **33099**, **79606**, **10104**, **14001** w/ division hash function and array size = 100.)
- Intuitive ideas regarding hashing, *collision*, and *collision resolution*
  - Hashing is inherently *deterministic*.
    - Hash function must yield the same hash value every time the same key is given.
      - But with different keys, hash values should be as well scattered as possible (within hash table range) – can we say *randomly deterministic*?
    - Whatever hash function and collision resolution strategy is used during storage, same must be used for access.
  - Hash function must be fast (easily and quickly computed) and why.
  - Collision is bad and why.
    - More collision -> more work during storage AND access.
    - No collision (perfect hashing) -> ideal, with collision -> deviates from ideal, increased collision -> decreased performance.

Factors affecting collision

- Hash table size
  - In general, increasing table size decreases collision.
    - Perfect hashing is an extreme case (table has a slot for every possible data item -> no collision).
  - Should be prime
    - For division type hash function, based on study by C. E. Radke: prime number that satisfies **`4k + 3`** (*e.g.*, 811)
  - Can *rehash* to adjust.
    - *Load factor* and performance monitoring.
- Hash function
  - *Uniform distribution* of probability over *entire range* of hash table to minimize collision (for given hash table size).
    - *E.g.* poor hash function: key = 10-uppercase-character string, 1000 hash table size, hash function sums up ASCII values of characters.
    - Why not use random-number generator as hash function?
      - Must be deterministically random.
      - Feasible with pseudo-random number generator (relatively slow to compute?) -> pseudo–random hashing.
    - 2 more types (involving numeric key) mentioned in textbook toward attaining deterministic randomness.
      - Mid-square hash function.
      - Multiplicative hash function.
- Nature of input.
- Collision resolution strategy.

Collision resolution strategy (policy, scheme) - quickly touch on the first three, focus on the fourth:

- **Overflow area**
- **Bucket size larger than one and overflow area**
- **Open hashing (chained hashing, separate chaining)**
- **Closed hashing (open addressing)**
  - Systematically computing a sequence of desirable alternative locations (*probe sequence*) within the hash table, starting with home location loc_0

    `loc₀ = h(key)`
  - Several commonly used strategies arise from following rather intuitive approach:

    As long as there's collision, try the next location in the sequence given by

    loc_i = (loc_0 + p(i, key)) % TableSize   (i = 1, 2, 3, ...)

    which, for the cases we'll discuss, can be thought of as

    loc_i = (loc_0 + i*StepSize) % TableSize   (i = 1, 2, 3, ...)
  - A desirable property we'd like `p(i, key)` (thus `StepSize`) to have: probe sequence should *cover entire hash table*.
  - If we choose `p(i, key) = i` (equivalently `StepSize = 1`), we get what is commonly called *linear probing*.
    - Essentially locate and use the next available hash table location (wrap around where necessary).
    - Main problem is *primary clustering* - show how the probability distribution changes as collisions happen.
  - One way to avoid primary clustering is to choose p(i, key) = i^2 (equivalently `StepSize = i`), called *quadratic probing*.
    - Probe sequence usually does not cover entire hash table (coverage may be as low as less than 50% if table size is not prime).
    - Suffers from *secondary clustering* - due to use of same sequence of step sizes when collisions happen, regardless of key).
  - Most common way to avoid primary clustering is to choose p(i, key) = i*h_2(key) (equivalently StepSize = h_2(key)), called *double hashing*.
    - h_2(key) should be carefully chosen so that values it provides for `StepSize` will cover entire hash table.
    - Intuitively, `StepSize` (which we'd want to be in range 1 through `TableSize - 1`) and `TableSize` should be *relatively prime*.
    - With above in mind and for the most commonly used division type hash function (when keys are integral numbers), Donald Knuth suggested the following possibility:
      - Have `TableSize` and `TableSize - 2` both be prime (called *twin primes*).
      - Use h_1(key) = key % TableSize.
      - Use h_2(key) = 1 + (key % (TableSize - 2)).
  - Example to illustrate linear probing and double hashing.
    - Last sample question (on hashing) of Sample Past Final Exam Questions.

More on hash functions (Lecture Note ***324s01AdditionalNotesOnHashFunctions***)

- To be candidate: (1) ***hash-value range*** must cover ***entire hash-table range***, (2) must be cheap/fast to calculate.
- To be winning candidate:

  (1) Determines hash value fully from key.
  (2) Uses entire key
  (3) Distributes keys uniformly
  (4) Gives very different hash values for similar keys
- Some "notable" hash functions for keys that are strings.