CODING BLOCKS

Lecture-17

Hashing

- Hashing Techniques
- Separate Chaining
- Linear Probing

Kartik Mathur

Any doubts?

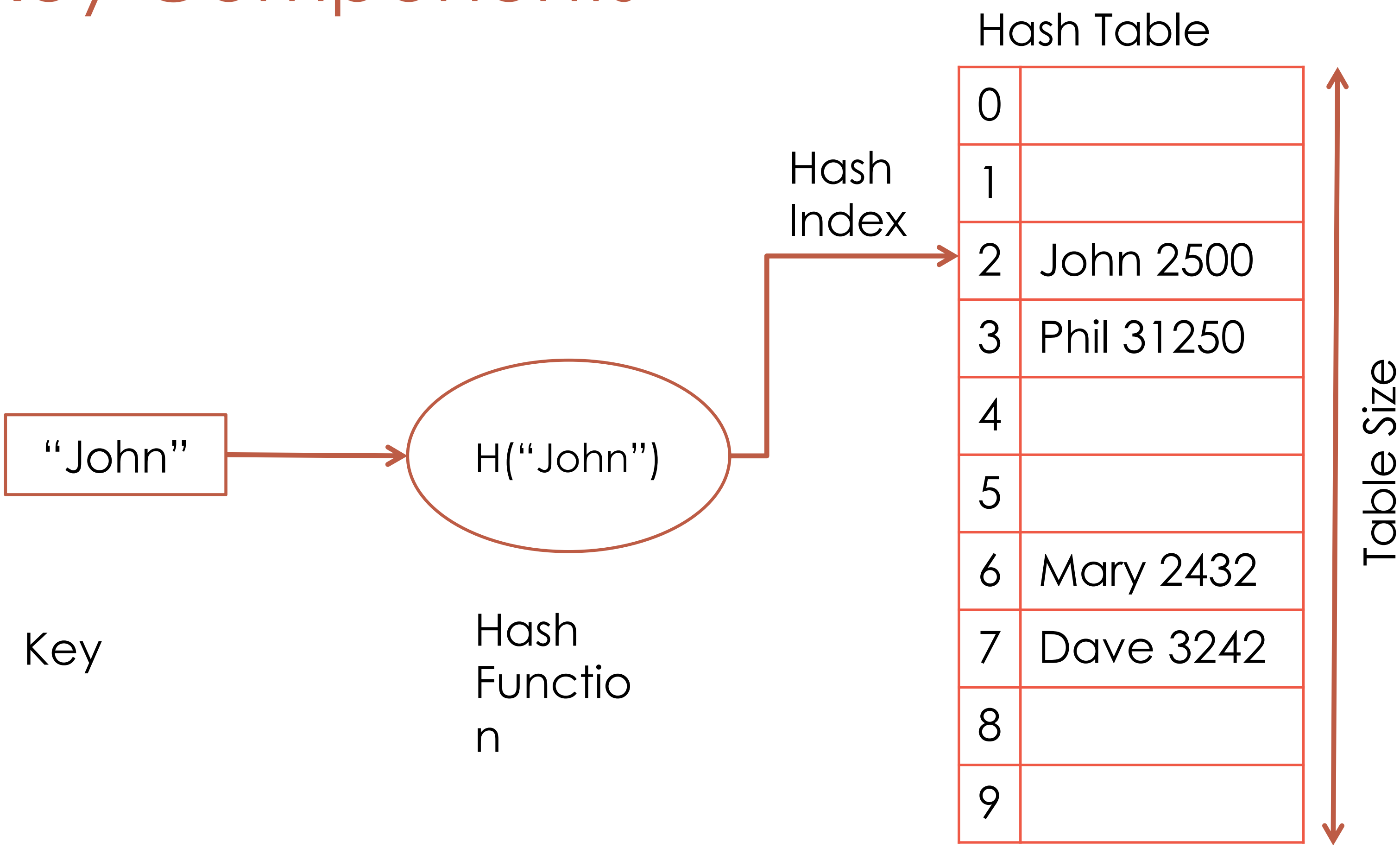
Overview

- Hash Table Data Structure : Purpose
  - To support insertion, deletion and search in average-case constant time
    - Assumption: Order of elements irrelevant
    - data structure *__not__* useful for if you want to maintain and retrieve some kind of an order of the elements
- Hash function
  - Hash[“string key”] ==> integer value

  *Value is a unique attribute

CODING BLOCKS

Key Components



How to determine *Hash Function* and *Table Size*?

Hash Table

- Hash table is an array of fixed size TableSize
- Array elements indexed by a key, which is mapped to an array index (0 to TableSize -1)
- Mapping (hash function) h from key to index
  - e.g., h("john") = 2

Hash Table Operations

- Insert – T[h(key)] = value;
- Delete – T[h(key)] = NULL;
- Search – return T[h(key)];

What happens if h("john") == h("joe")

Collision!

CODING BLOCKS

Factors!

- Hash Function
- Table Size – usually fixed at the beginning
- Collision handling Scheme

Hash Function

h(key) => hash table index

- Collisions cannot be avoided but its chances can be reduced using a "good" hash function.
- A "good" hash function should have the properties:
  - Reduced chance of collision - Distribute keys uniformly over table
  - Should be fast to compute

Effective use of Table Size

- Simple hash function (assume integer keys)

  h(Key) = Key % TableSize

- For random keys, h() distributes keys evenly over table
  - What if TableSize = 100 and keys are ALL multiples of 10?
  - Better if TableSize is a prime number

What about strings?

- Add up character ASCII values (0-255) to produce integer keys
  - E.g., “abcd” = 97+98+99+100 = 394
  - h(“abcd”) = 394 % TableSize
- Potential problems:
  - Anagrams will map to the same index [h(“abcd”) == h(“dbac”)]
  - Small strings may not use all of table –
    [Strlen(S) * 255 < TableSize]
  - Time proportional to length of the string

So lets try something else?

- Treat first 3 characters of string as base-27 integer (26 letters plus space)

$$Key = S[0] + (27 * S[1]) + (27^2 * S[2])$$

- Potential problems:
  - Assumes first 3 characters randomly distributed – Not true English

Another attempt!

- Use all N characters of string as an N-digit base-K number
- Choose K to be prime number larger than number of different digits (characters). i.e k = 29, 31, 37 etc.

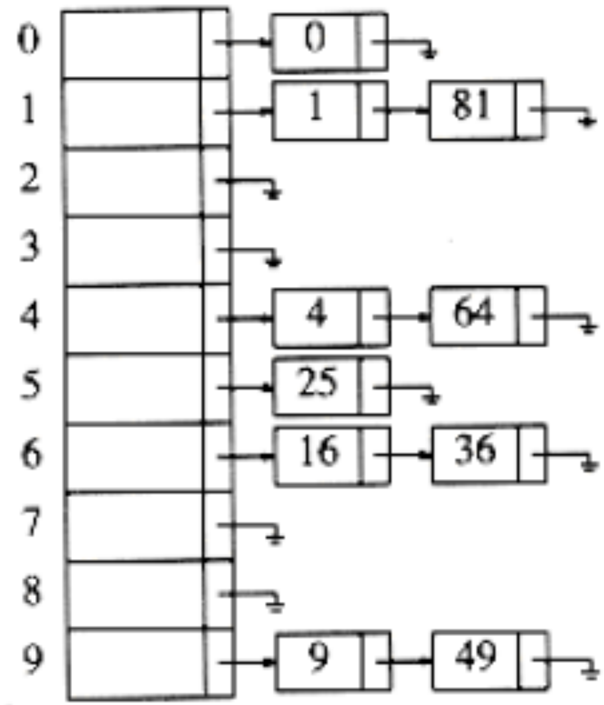
$$h(S) = \left[\sum_{i=0}^{L-1} S[L-i-1] * 37^i \right] \mathrm{mod}\, TableSize$$

How to handle Collisions?

- Open Hashing – Separate Chaining
- Closed Hashing – Open Addressing
  - Linear Probing
  - Quadratic Probing
- Double Hashing

Separate Chaining

- Implemented using Linked Lists.
- Key k is stored in list at T[h(k)]
- E.g., TableSize = 10
  - h(k) = k mod 10
  - Insert first 10 perfect squares



Lets see implementation!

Disadvantages

- Linked lists could get long which impacts performance
- More memory because of pointers
- Absolute worst-case (even if N << M)
  - All N elements in one linked list!
  - Typically the result of a bad hash function

Open Addressing

When a collision occurs, look elsewhere in the table for an empty slot.

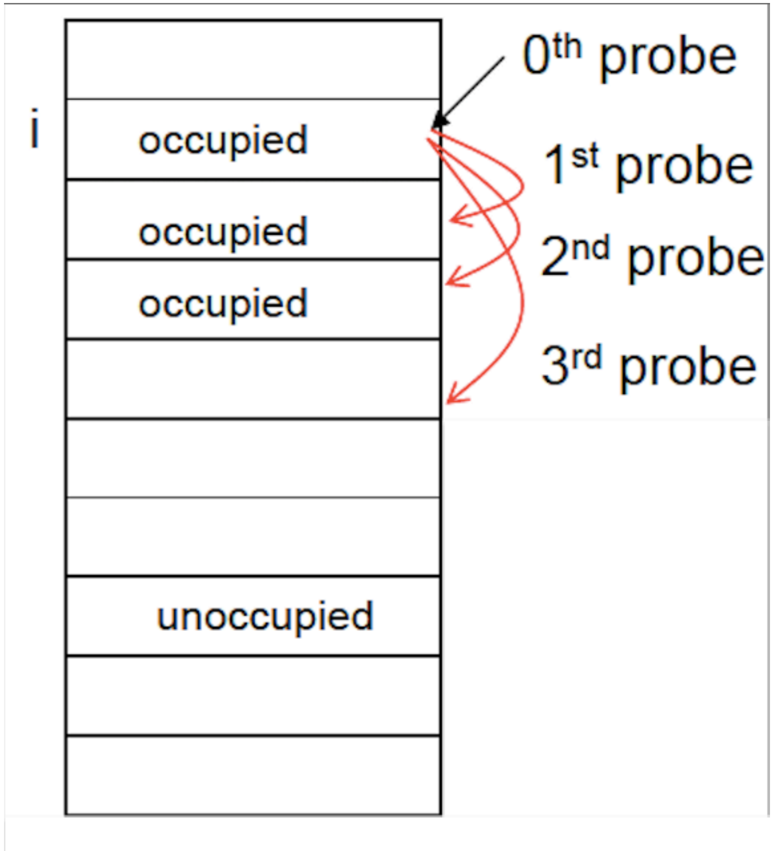
- Advantages over chaining
  - No need for list structures
  - No need to allocate/deallocate memory during insertion/deletion (slow)
- Disadvantages
  - Slower insertion – May need several attempts to find an empty slot
  - Table needs to be bigger (than chaining-based table) to achieve average-case constant-time performance

Probe Sequence

- A “Probe sequence” is a sequence of slots in hash table while searching for an element
  - h0(x), h1(x), h2(x), …
  - Needs to visit each slot exactly once
  - Needs to be repeatable (so we can find/delete what we’ve inserted)
- Hash Function
  - hi(x) = (h(x) + f(i)) mod TableSize
  - f(0) = 0
  - f is the collision resolution strategy

Linear Probing

f(i) = is a linear function of i e.g., f(i) = i



Quadratic Probing

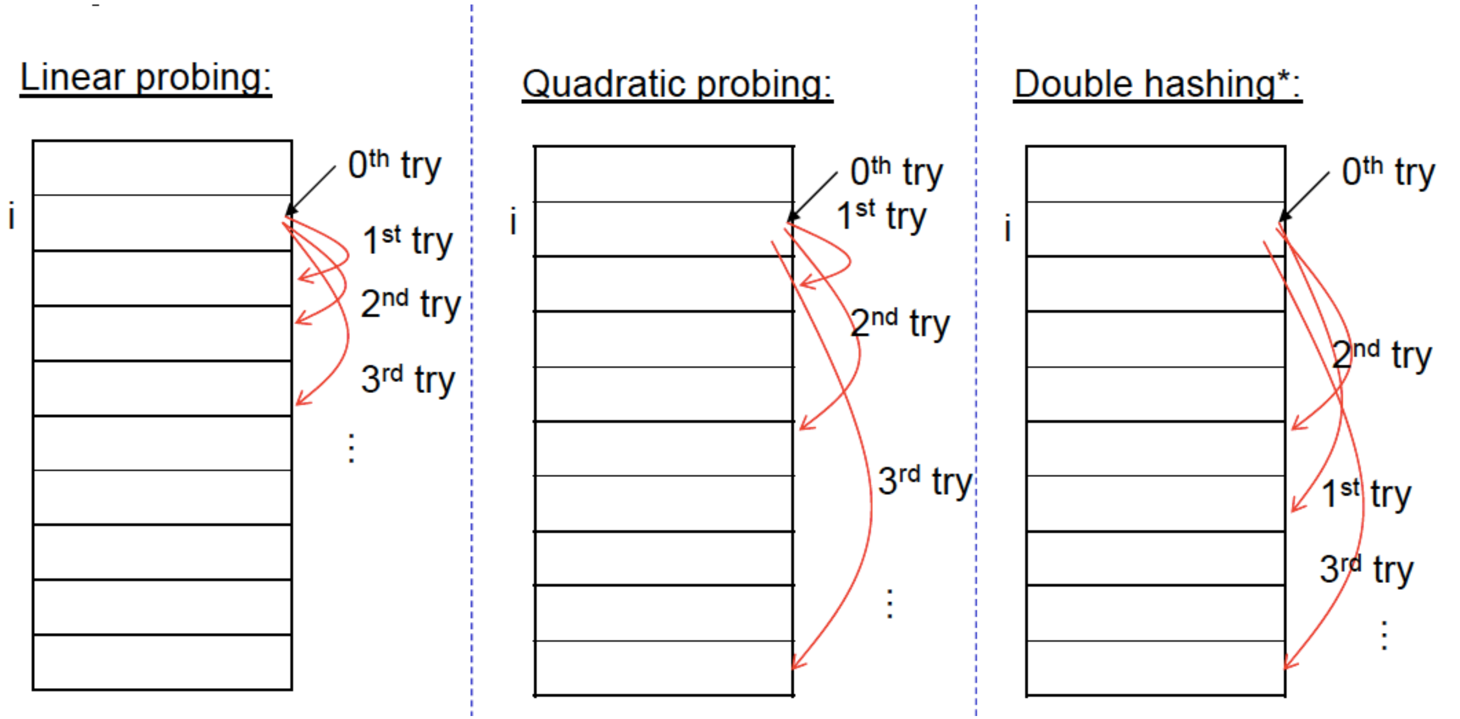
- Avoids primary clustering
- f(i) is quadratic in i - $f(i)= i^2$
  - Theorem – New element can always be inserted into a table that is at least half empty and TableSize is prime
  - Otherwise, may never find an empty slot, even is one exists
  - Ensure table never gets half full. If close, then expand it

CODING BLOCKS

Double Hashing

- Use a second hash function for all tries of i other than 0: f(i) = i * h2(x)
- Good choices for h2(x) ?
  - Should never evaluate to 0
  - h2(x) = R – (x mod R), where R is prime number less than TableSize

Probing Techniques - review



Load Factor

Load factor λ of a hash table T is defined as follows:

- N = number of elements in T ("current size")
- M = size of T ("table size")
- $\lambda = N/M$ (" load factor")

- If the load factor is kept reasonable, the hash table should perform well, provided the hashing is good.
- If the load factor grows too large, the hash table will become slow, or it may fail to work (depending on the method used).
- For a fixed number of buckets, the time for a lookup grows with the number of entries and so does not achieve the desired constant time.
- Ideally we want $\lambda <= 1$, Not a function of N.

CODING BLOCKS

Rehashing

- Increases the size of the hash table when load factor becomes "too high" (defined by a cutoff)
  - Anticipating that prob(collisions) would become higher
- Typically expand the table to twice its size (but still prime)
- Need to reinsert all existing elements into new hash table

CODING BLOCKS

Thank You!

Kartik Mathur