C++

LAUNCHPAD

CODING BLOCKS

Lecture-20

Hashing

- Hashing Techniques
- Separate Chaining
- Linear Probing

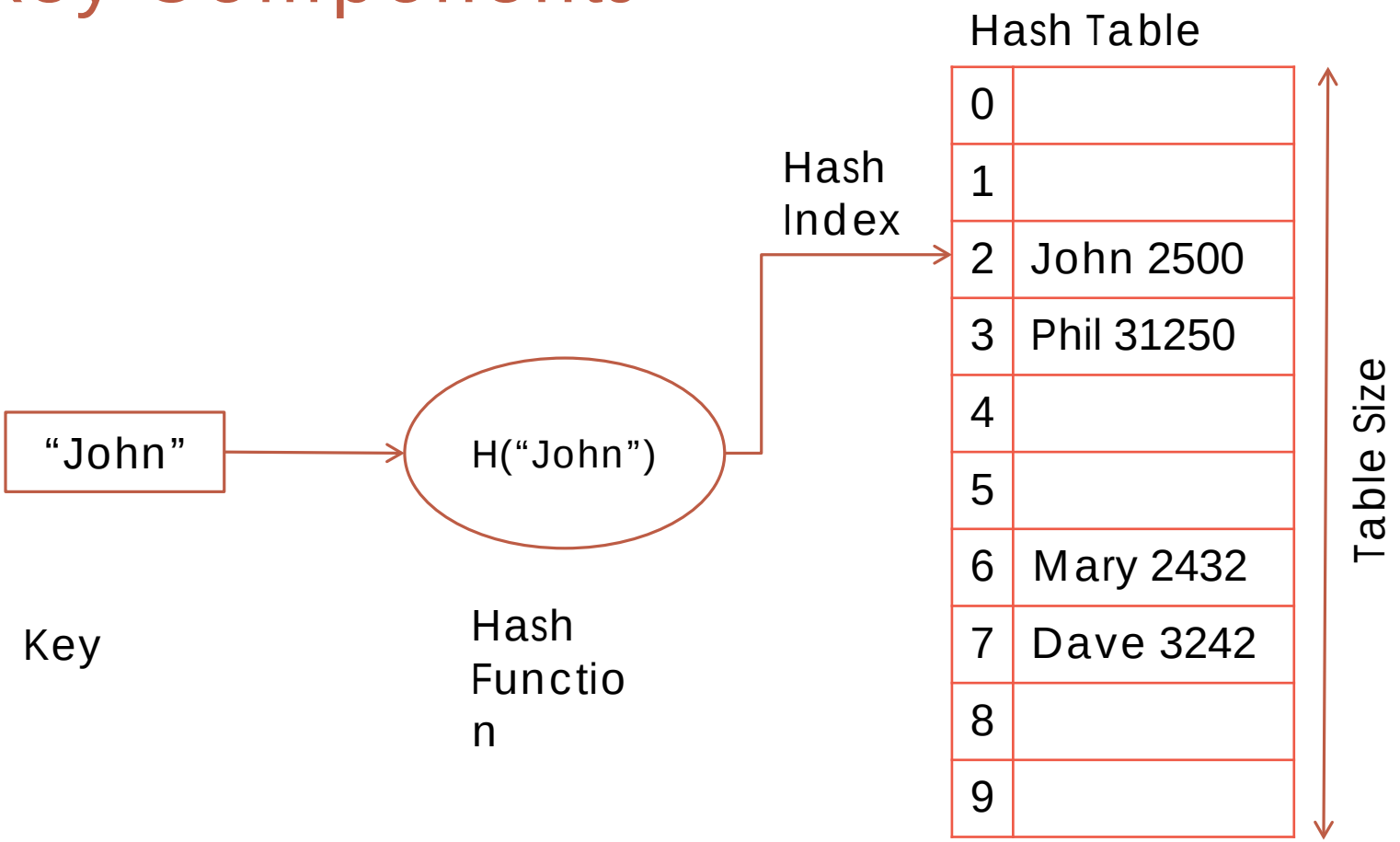
Prateek Narang

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

Key Components



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

CODING BLOCKS

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!

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.

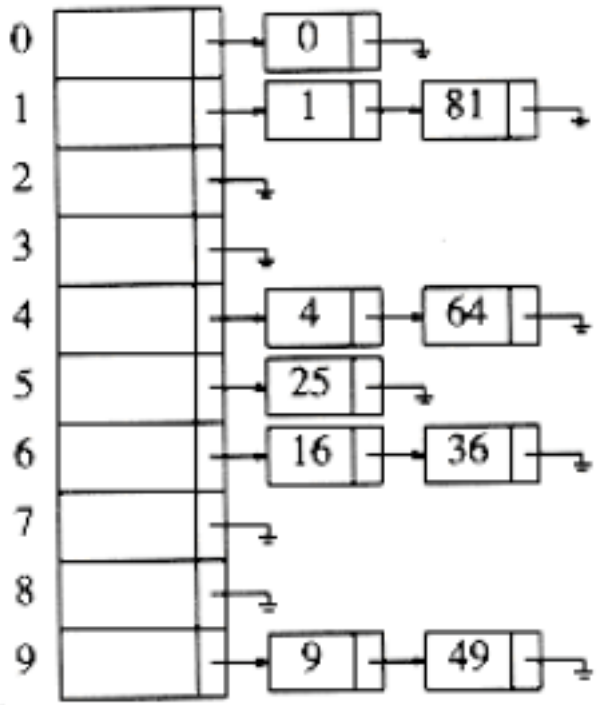
$$h(S) = \left[\sum_{i=0}^{L-1} S[L-i-1] * 37^i \right] \bmod TableSize$$

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



CODING BLOCKS

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.

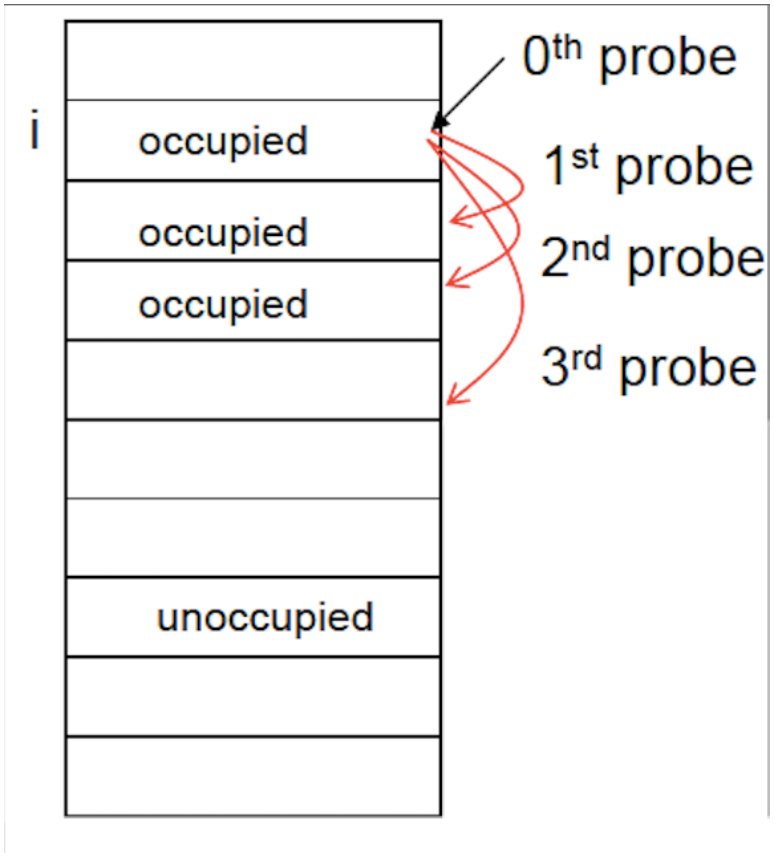
- Advantages over chaining
  - No need for list structures
  - No need to allocate/deallocate memory during insertion/deletion (slow)
- Disadvantages
  - Slower insertion – May need several attempts to find an empty slot
  - Table needs to be bigger (than chaining-based table) to achieve average-case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



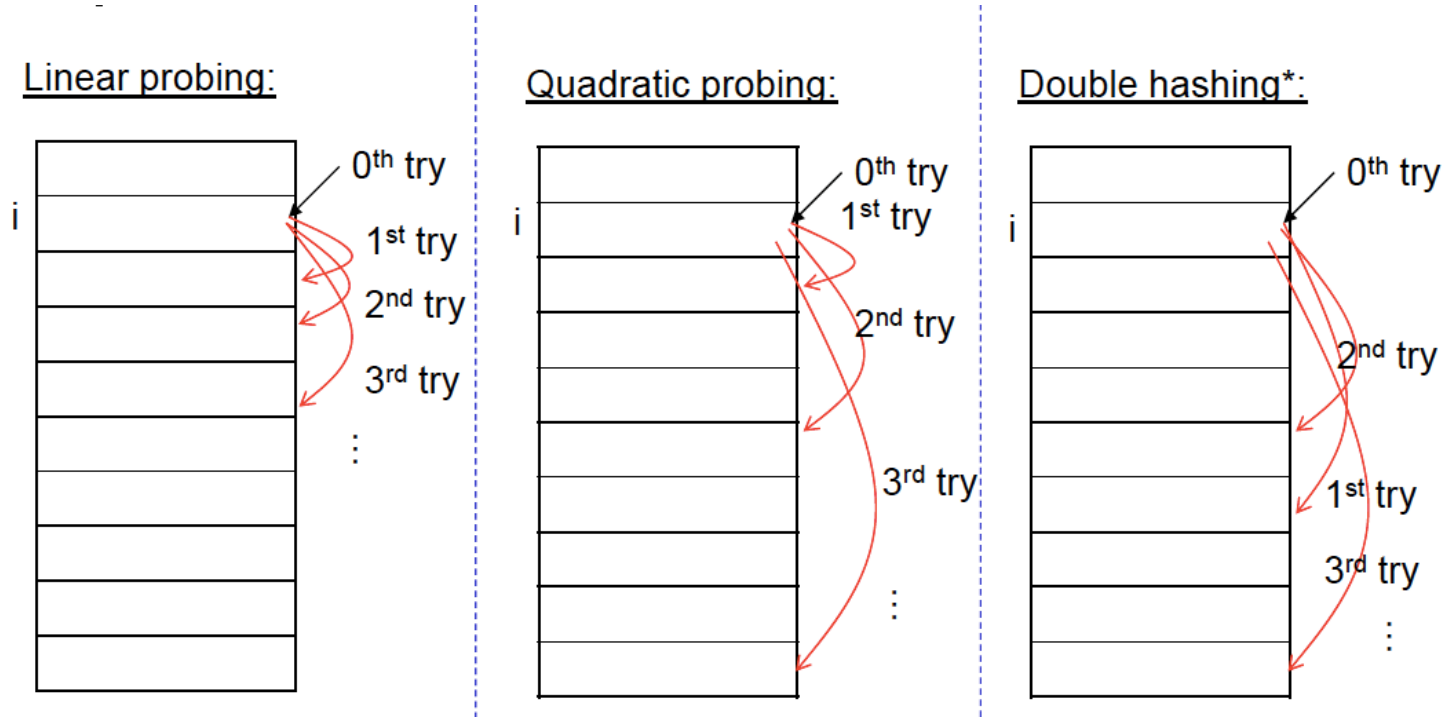
Quadratic Probing

- 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

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



CODING BLOCKS

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”)
- λ = 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 λ <= 1, Not a function of N.

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

CPP Implementation

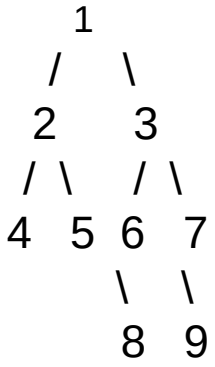
- Unordered Set [unordered_set]
- Unordered Map[unordered_map]-
  Separate Chaining
- Map [map]
  - Self-Balancing BST / Red-Black Tree

They were incorporated into the C++11 revision of the C++ standard.

Implement Quadratic Probing yourself!

Lets discuss some problems

- Find intersection of two sorted arrays. What about unsorted arrays?
- Print a Binary Tree in Vertical Order



The output of print this tree vertically will be:

4
2
1 5 6
3 8
7
9

CODING BLOCKS

Time to try?

- Find pair of elements which sum to zero.
- Union and Intersection of two Linked Lists
- Remove duplicates from arrays.

C++ Vectors

#include<vector>

- Just like arrays, vectors use contiguous storage locations for their elements, which means that their elements can also be accessed using offsets on regular pointers to its elements, and just as efficiently as in arrays.
- But unlike arrays, their size can change dynamically, with their storage being handled automatically by the container

Common Operations

- push_back
- pop_back
- clear
- empty
- insert
- erase
- []
- at
- front
- back

C++ Lists

#include<list>

List containers are implemented as doubly-linked lists; Doubly linked lists can store each of the elements they contain in different and unrelated storage locations. The ordering is kept internally by the association to each element of a link to the element preceding it and a link to the element following it

Common Operations

- push_back
- pop_back
- push_front
- pop_front
- clear
- empty
- Insert
- erase
- front
- back
- size

STL Reference & Tutorials

- http://www.cplusplus.com/reference/stl/
- http://www.codeguru.com/cpp/cpp/cpp_mfc/stl/article.php/c4027/C-Tutorial-A-Beginners-Guide-to-stdvector-Part-1.htm
- http://www.mochima.com/tutorials/vectors.html

Misc Problems on Recursion

- Rat in a Maze
- Knights Problem
- Sudoku Solver

C++

LAUNCHPAD

CODING BLOCKS

Thank You!

Prateek Narang