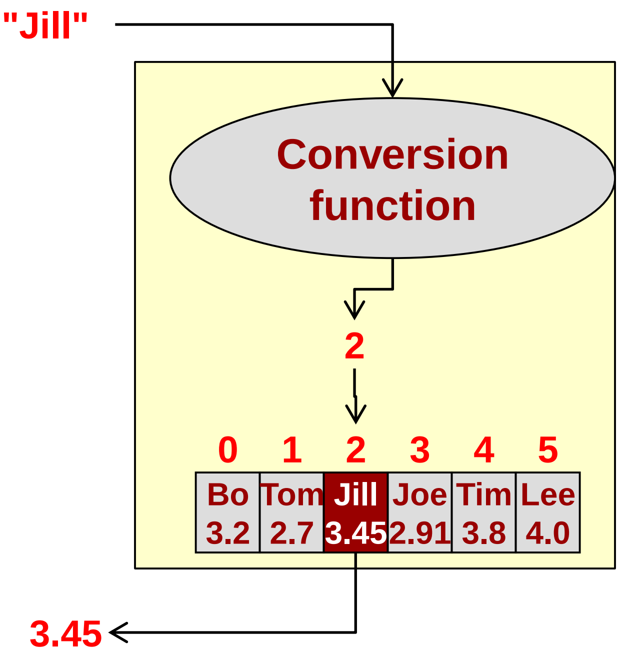
CSCI 104
Hash Tables & Functions

Mark Redekopp
David Kempe
Sandra Batista

Unordered_Maps / Hash Tables

- A hash table implements a map ADT
  - Add(key,value)
  - Remove(key)
  - Lookup/Find(key) : returns value
- In a BST the keys are kept in order
  - A Binary Search Tree implements an **ORDERED MAP**
- In a hash table keys are evenly distributed throughout the table (unordered)
  - A hash table implements an **UNORDERED MAP**

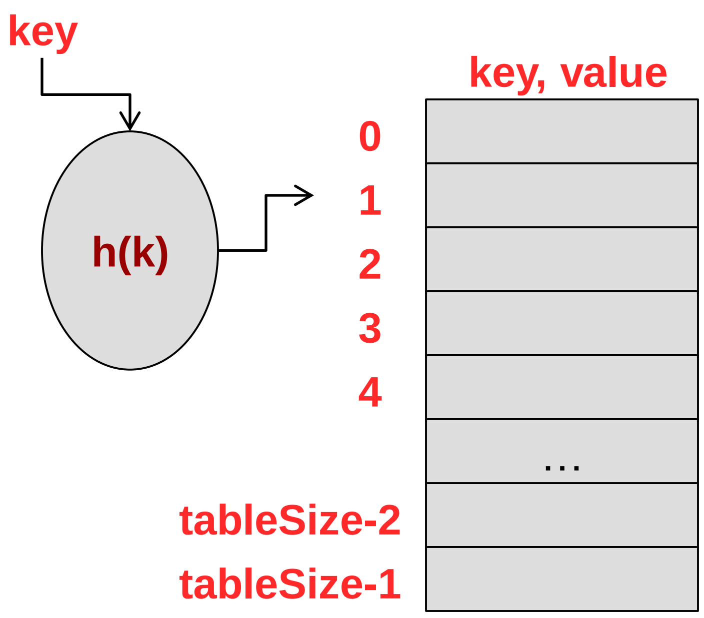


C++11 Implementation

- C++11 added new container classes:
  - `unordered_map`
  - `unordered_set`
- Each uses a hash table for average complexity to insert , erase, and find in O(1)
- Must compile with the `-std=c++11` option in g++

Hash Tables

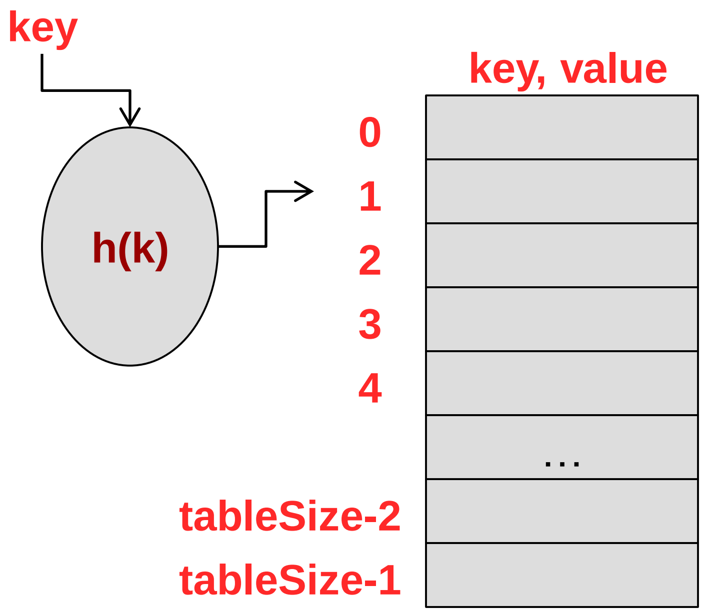
- A hash table is an array that stores key,value pairs
  - Usually smaller than the size of possible set of keys,
- The table is coupled with a **hash function**, *h(k)*, that maps keys to an integer in the range [0..tableSize-1] (i.e. [0 to **m**-1])
- The ***hash function, h(k)*** should be fast to compute (O(1))



m = tableSize
n = # of keys entered

General Table Size Guidelines

- The table size should be bigger than the amount of expected entries (**m** > **n**)
- TableSize should usually be a **prime number**



m = tableSize
n = # of keys entered

Hash Functions First Look

- Challenge: Distribute keys to locations in hash table such that
- **Easy to compute** and retrieve values given key
- Keys **evenly distributed** throughout the table
- Distribution is **consistent for retrieval**
- If necessary, key data type is converted to integer before hash is applied

Hash Function Goals

- Common Hash Function: **h(k) = k mod m**

where m is prime table size

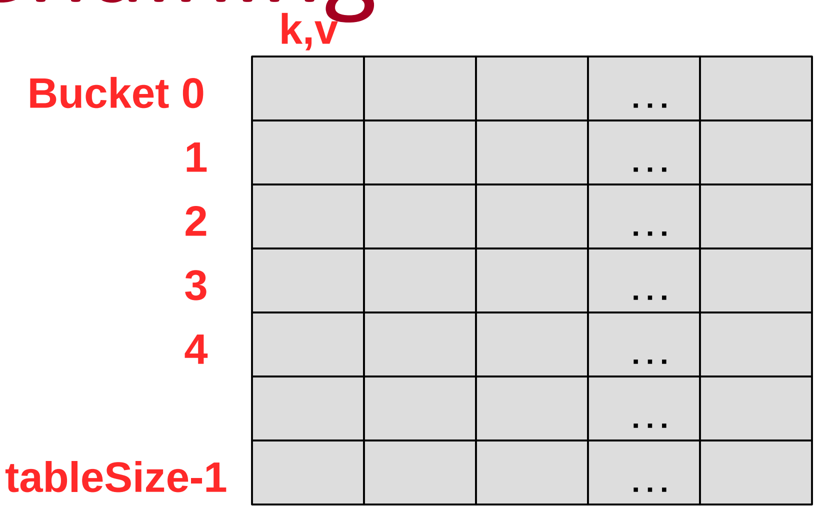
- Rules of thumb
  - The hash function should examine the entire search key, not just a few digits or a portion of the key
  - When modulo hashing is used, the base should be prime
- A "perfect hash function" should map each of the **n** keys to a unique location in the table
- A "good" hash function or *Universal Hash Function*
  - P(h(k) = x) = 1/**m** (i.e. **pseudorandom**)

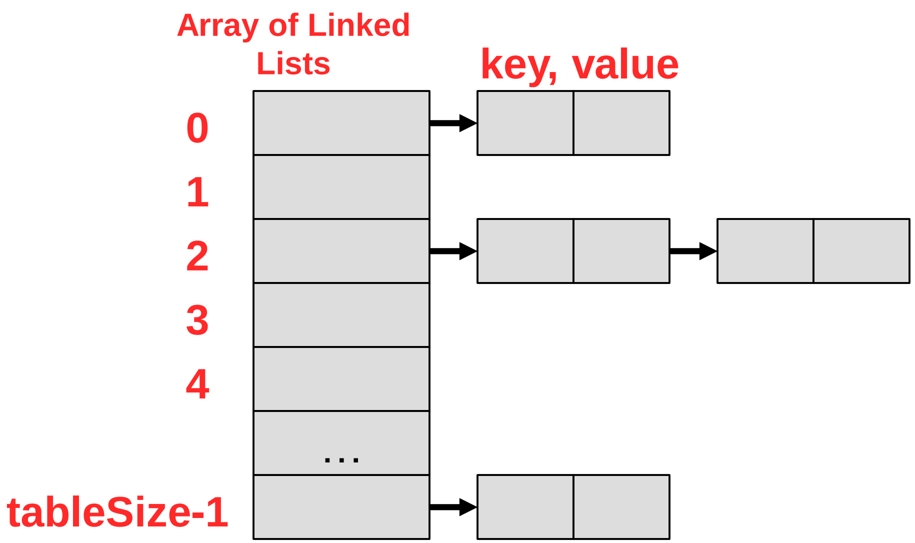
Resolving Collisions

- Collisions occur when two keys, k1 and k2, are not equal, but h(k1) = h(k2).
- Collisions are inevitable if the number of entries, **n**, is greater than table size, **m** (***by pigeonhole principle***)
- Methods
  - Closed Addressing (e.g. buckets or **chaining**)
  - Open addressing (aka probing)
    - Linear Probing
    - Quadratic Probing
    - Double-hashing

Buckets/Chaining

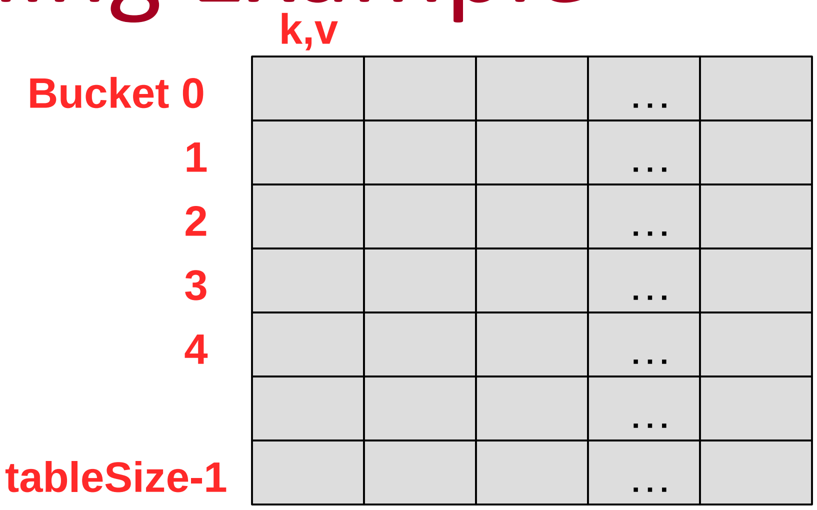
- Rather than searching for a free entry, make each entry in the table an ARRAY (bucket) or LINKED LIST (chain) of items/entries
- Buckets
  - How big should you make each
  - array?
  - Too much wasted space
- Chaining
  - Each entry is a linked List

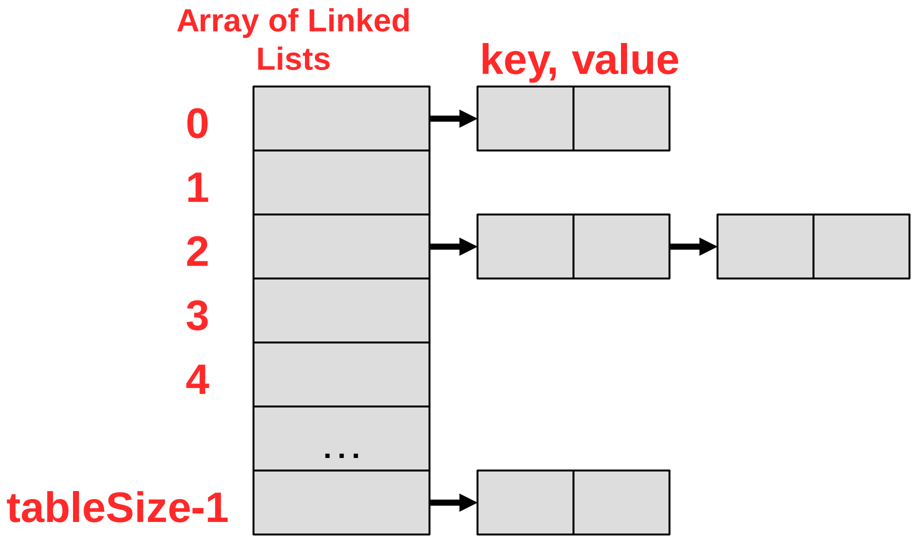




Buckets/Chaining Example

- Let h(k) = k mod 17
- Table size m = 17
- Insert keys 34, 19, 36

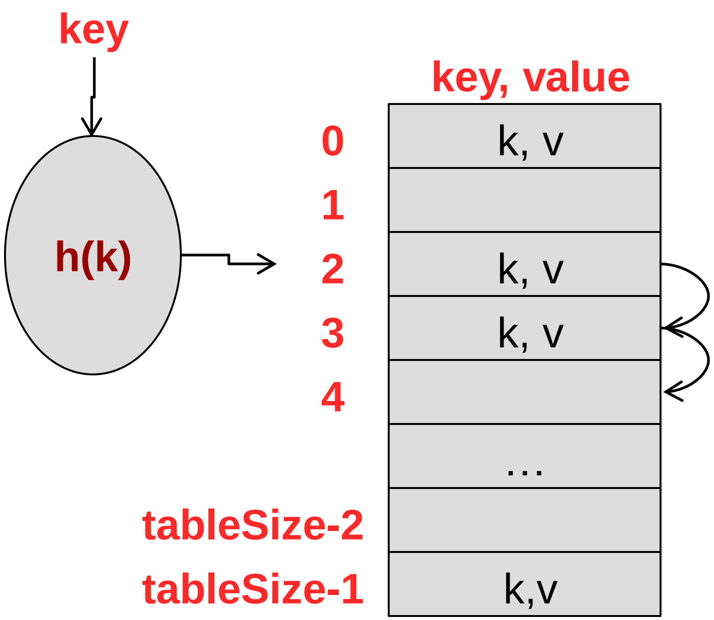




Open Addressing

- Open addressing means an item with key, k, may not be located at h(k)

- Let i be number of failed inserts
- Linear Probing
  - h(k,i) = (h(k)+i) mod m
  - Example: Check h(k)+1, h(k)+2, h(k)+3, …
- Quadratic Probing
  - h(k,i) = (h(k)+i^2) mod m
  - Check location $h(k)+1^2$, $h(k)+2^2$, $h(k)+3^2$, …



Linear Probing Issues

- Linear probing leads to clusters of occupied areas in the table called ***primary clustering***
- How would quadratic probing help fight primary clustering?
  - Quadratic probing tends to spread out data across the table.

	key, value
0	occupied
1	
2	occupied
3	occupied
4	
	…
tableSize-2	
tableSize-1	occupied

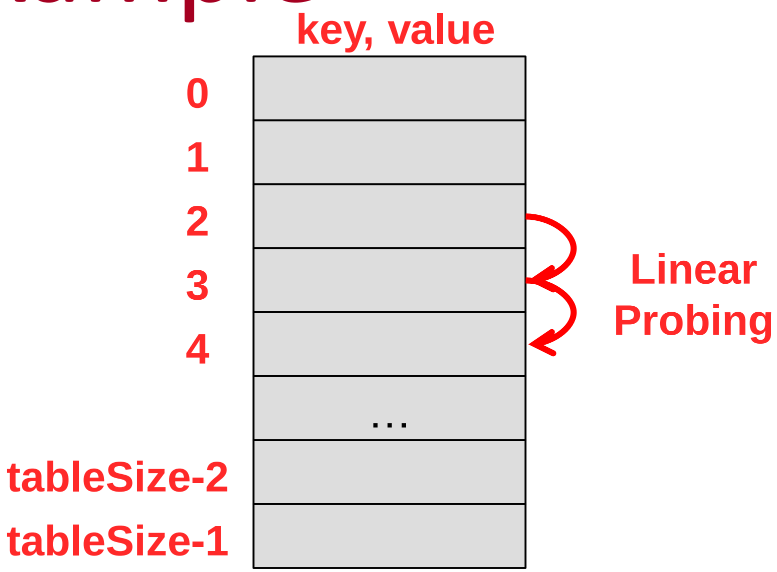
Linear Probing

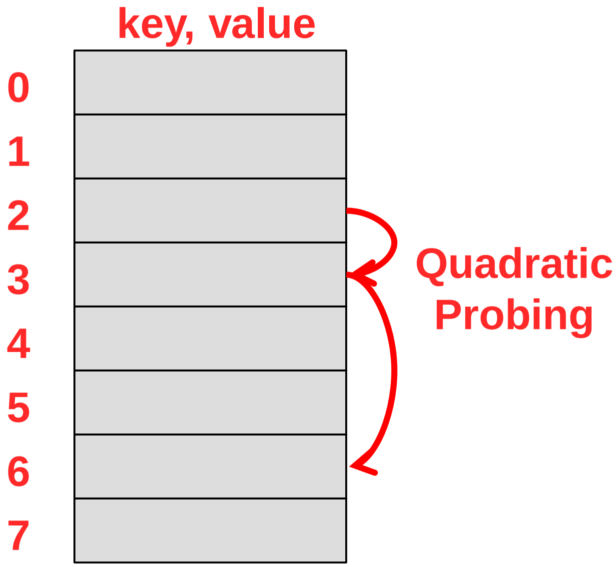
	key, value
0	occupied
1	
2	occupied
3	occupied
4	
5	
6	
7	occupied

Quadratic Probing

Clustering Example

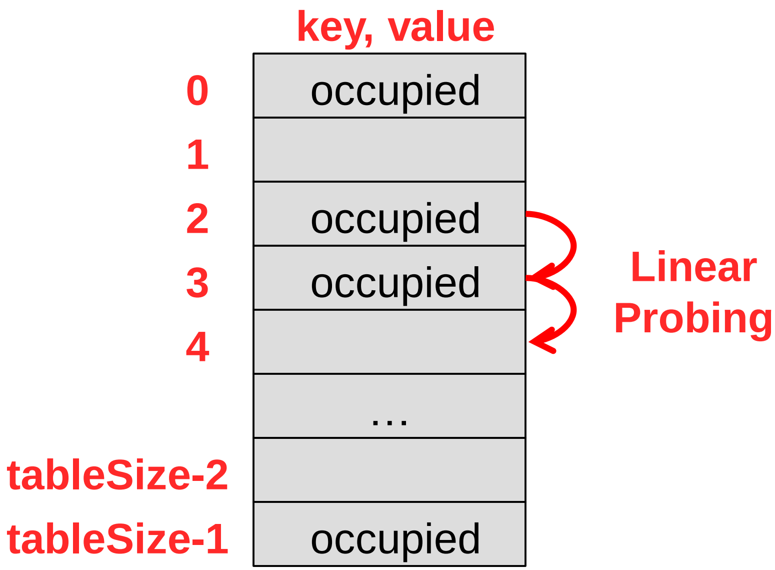
- Let h(k) = k mod 17
- Table size m = 17
- Insert keys 19, 36, 2
- Use linear probing and then quadratic probing

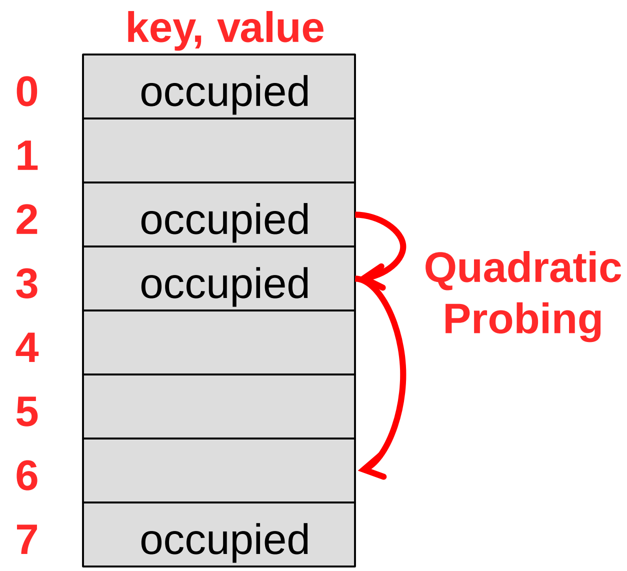




Find, find(k)

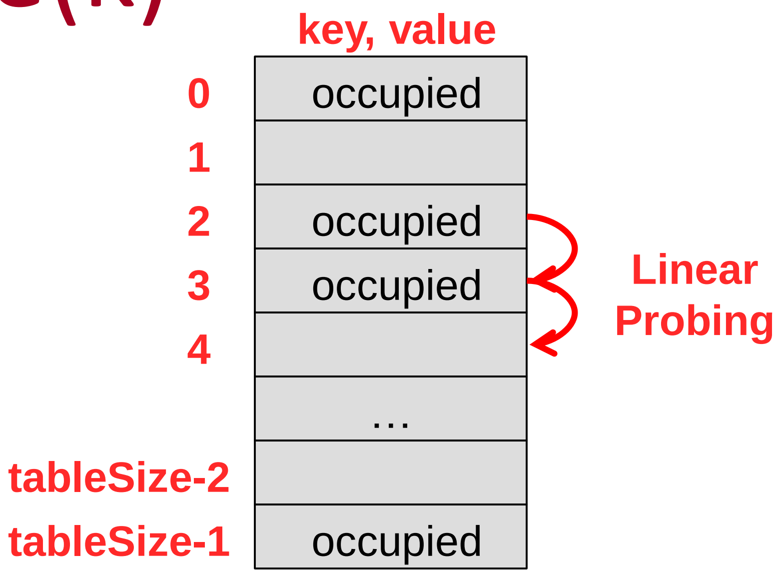
- Given open addressing scheme how would you find a given key, value pair
  - First hash it
  - If it is not at h(k), follow probing sequence until
    - Find key -> return found
    - Find an empty -> return not found
    - Search the whole table -> return not found

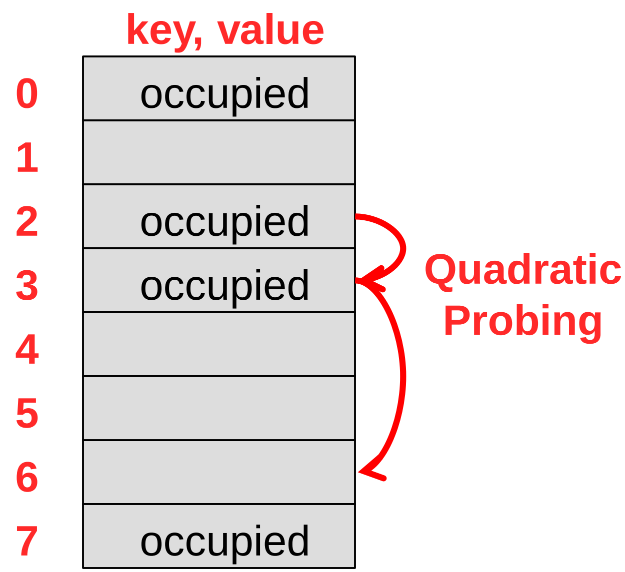




Removal, remove(k)

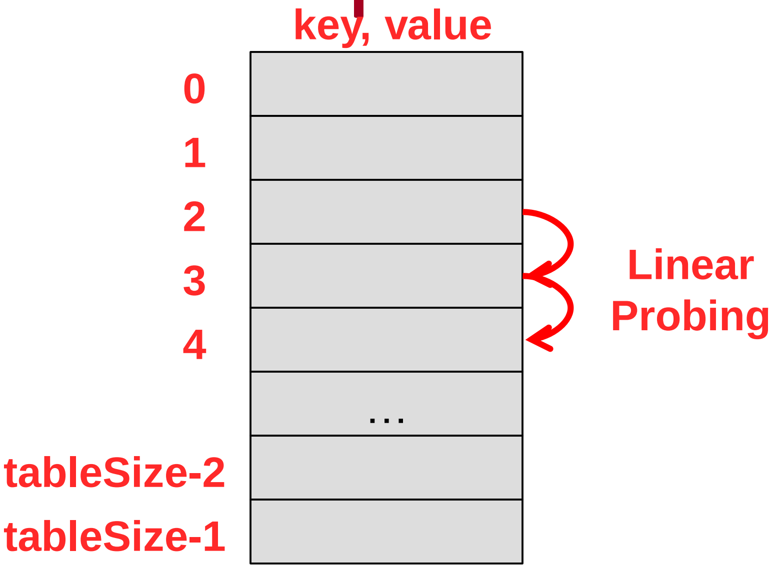
- If closed addressing such as chaining, find k and delete it
- If open addressing such as linear or quadratic probing:
  - find(k)
  - Mark a location as "removed"=unoccupied but part of a cluster





Find and Remove Example

- Let h(k) = k mod 17
- Table size m = 17
- Use linear probing
- Insert keys 19, 36, 2
- Remove 36, Find 2



Double Hashing

- Define $h_1(k)$ to map keys to a table location
- But also define $h_2(k)$ to produce a linear probing step size
  - First look at $h_1(k)$
  - Then if it is occupied, look at $h_1(k) + h_2(k)$
  - Then if it is occupied, look at $h_1(k) + 2*h_2(k)$
  - Then if it is occupied, look at $h_1(k) + 3*h_2(k)$
- TableSize=13, h1(k) = k mod 13, and h2(k) = 5 – (k mod 5)
- What sequence would I probe if k = 31
  - h1(31) = ___, h2(31) = _______________
  - Seq: ____

Double Hashing

- Define $h_1(k)$ to map keys to a table location
- But also define $h_2(k)$ to produce a linear probing step size
  - First look at $h_1(k)$
  - Then if it is occupied, look at $h_1(k) + h_2(k)$
  - Then if it is occupied, look at $h_1(k) + 2*h_2(k)$
  - Then if it is occupied, look at $h_1(k) + 3*h_2(k)$
- TableSize=13, h1(k) = k mod 13, and h2(k) = 5 – (k mod 5)
- What sequence would I probe if k = 31
  - h1(31) = 5, h2(31) = 5-(31 mod 5) = 4
  - 5, 9, 0, 4, 8, 12, 3, 7, 11, 2, 6, 10, 1

Practice

- Use the hash function h(k)=k%7 to find the contents of a hash table (m=7) after inserting keys 14, 8, 21, 2, 7 using double hashing. Let the second function be h2(k) = 3 - (k%3).

Hash Tables

- Suboperations
  – Compute h(k) should be O(1)
  – Array access of table[h(k)] = O(1)
- In a hash table, what is the expected efficiency of each operation
  – Find = O(1)
  – Insert = O(1)
  – Remove = O(1)

Hashing Efficiency

- Loading factor, α, defined as:
  - (n=number of items in the table) / m=tableSize => $\alpha = n / m$
  - Really it is just the fraction of locations currently occupied
- For chaining, α, can be greater than 1
  - The load factor is average length of chain
- Best to keep the loading factor, $\alpha < .5$ especially for probing
- Resize and rehash contents if load factor too large: (using new hash function):
  - Allocate larger table size
  - <u>Must rehash keys</u> to location in new table size.

Summary

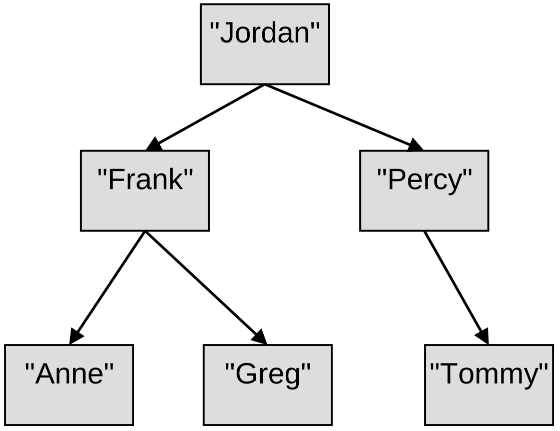
- Hash tables are LARGE arrays with a function that attempts to compute an index from the key
- In the general case, **chaining** is the best
- collision resolution approach
- The functions should spread the possible keys evenly over the table [i.e. $p(\ h(k) = x\ ) = 1/m$]

An imperfect set…

BLOOM FILTERS

Set Review

- Recall the operations a set performs…
  - Insert(key)
  - Remove(key)
  - Contains(key) : bool   (a.k.a. find())
- We can implement a set using
  - List
    - O(n) for some of the three operations
  - (Balanced) Binary Search Tree
    - O(log n) insert/remove/contains
  - Hash table
    - O(1) insert/remove/contains



Bloom Filter Idea

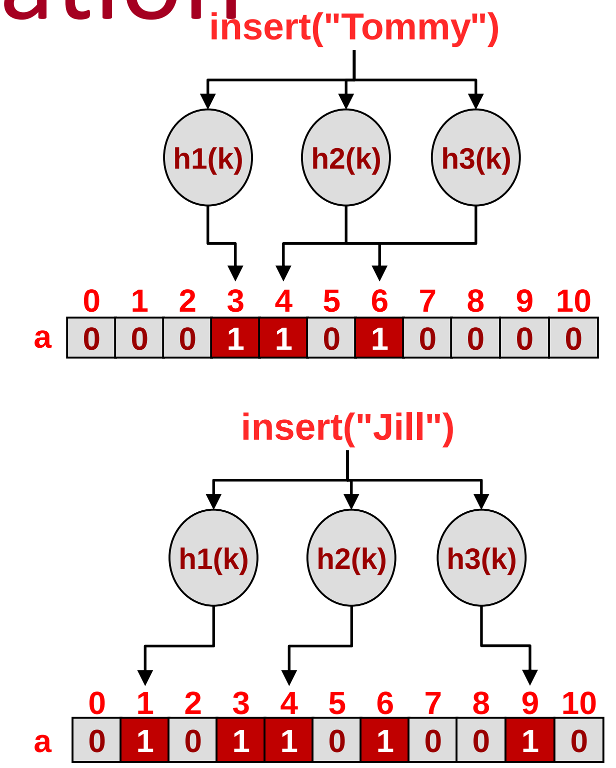
- A Bloom filter is a set such that "contains()" will *quickly* answer…
  - "No" correctly (i.e. if the key is not present)
  - "Yes" with a chance of being incorrect (i.e. the key may not be present but it might still say "yes")
- Why would we want this?

Bloom Filter Motivation

- Why would we want this?
  - A Bloom filter usually sits in front of an actual set/map
  - Suppose that set/map is EXPENSIVE to access
    - if set/map doesn't sits on a disk drive or another server
      - Disk/Network access = ~milliseconds
      - Memory access = ~nanoseconds
  - The Bloom filter is small enough to reside in memory for quick access and can answer quickly if the set/map on disk contains a key:
    - If it answers "No" do not search the EXPENSIVE set
    - If it answers "Yes" search the EXPENSIVE set

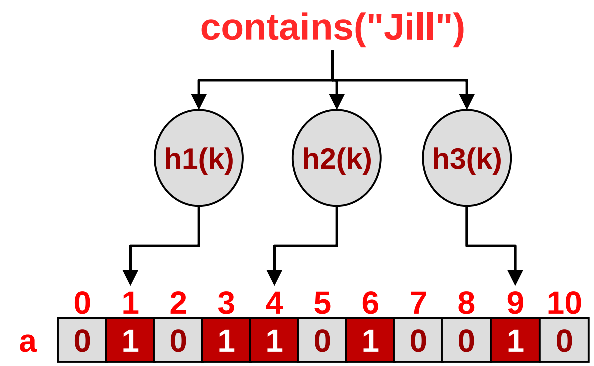
Bloom Filter Explanation

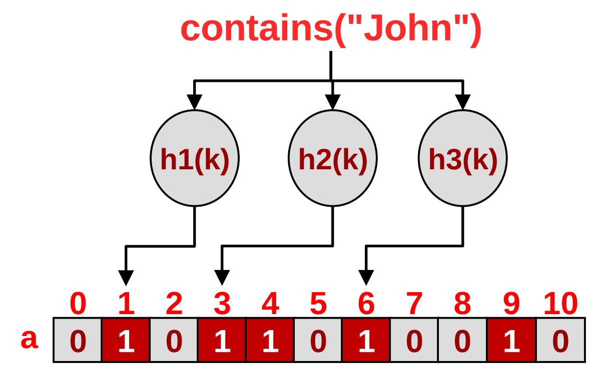
- A Bloom filter is…
  - A hash table of individual bits (Booleans: T/F)
  - A set of hash functions, $\{h_1(k), h_2(k), \ldots h_s(k)\}$
- Insert()
  - Apply each $h_i(k)$ to the key
  - Set $a[h_i(k)]$ = True



Bloom Filter Explanation

- A Bloom filter is…
  - A hash table of individual bits (Booleans: T/F)
  - A set of hash functions, $\{h_1(k), h_2(k), \ldots h_s(k)\}$
- Contains()
  - Apply each $h_i(k)$ to the key
  - Return True if ***all*** $a[h_i(k)]$ = True
  - Return False otherwise
  - In other words, answer is "Maybe" or "No"
    - May produce "false positives"
    - May NOT produce "false negatives"
- We will ignore removal for now





Practice

- Trace a Bloom Filter on the following operations:
  - insert(0), insert(1), insert(2), insert(8), contains(2), contains(3), contains(4), contains(9)
  - The hash functions are
    - h1(k)=(7k+4)%10
    - h2(k) = (2k+1)%10
    - h3(k) = (5k+3)%10
    - The table size is 10 (m=10).

	0	1	2	3	4	5	6	7	8	9
a	0	0	0	0	0	0	0	0	0	0

	H1(k)	H2(k)	H3(k)	Hit?
Insert(0)	4	1	3	N/A
Insert(1)	1	3	8	N/A
Insert(2)	8	5	3	N/A
Insert(8)	0	7	3	N/A
Contains(2)	8	5	3	Yes
Contains(3)	5	7	8	Yes
Contains(4)	2	9	3	No
Contains(9)	7	9	8	No