# Lecture 13

Christopher Godley

CSCI 2270 Data Structures

July 23rd, 2018

# Hash Tables

- A hash table stores data using a *key*
  - This *key* is a unique identifier for the data, but it maps into a discrete record location
- The data may be referred to as a *record,* and the array that stores the records is called the *hash table*
- A hash table has two necessary components:
  - The array where records are stored
  - A **hash function** that maps a *key* to an index

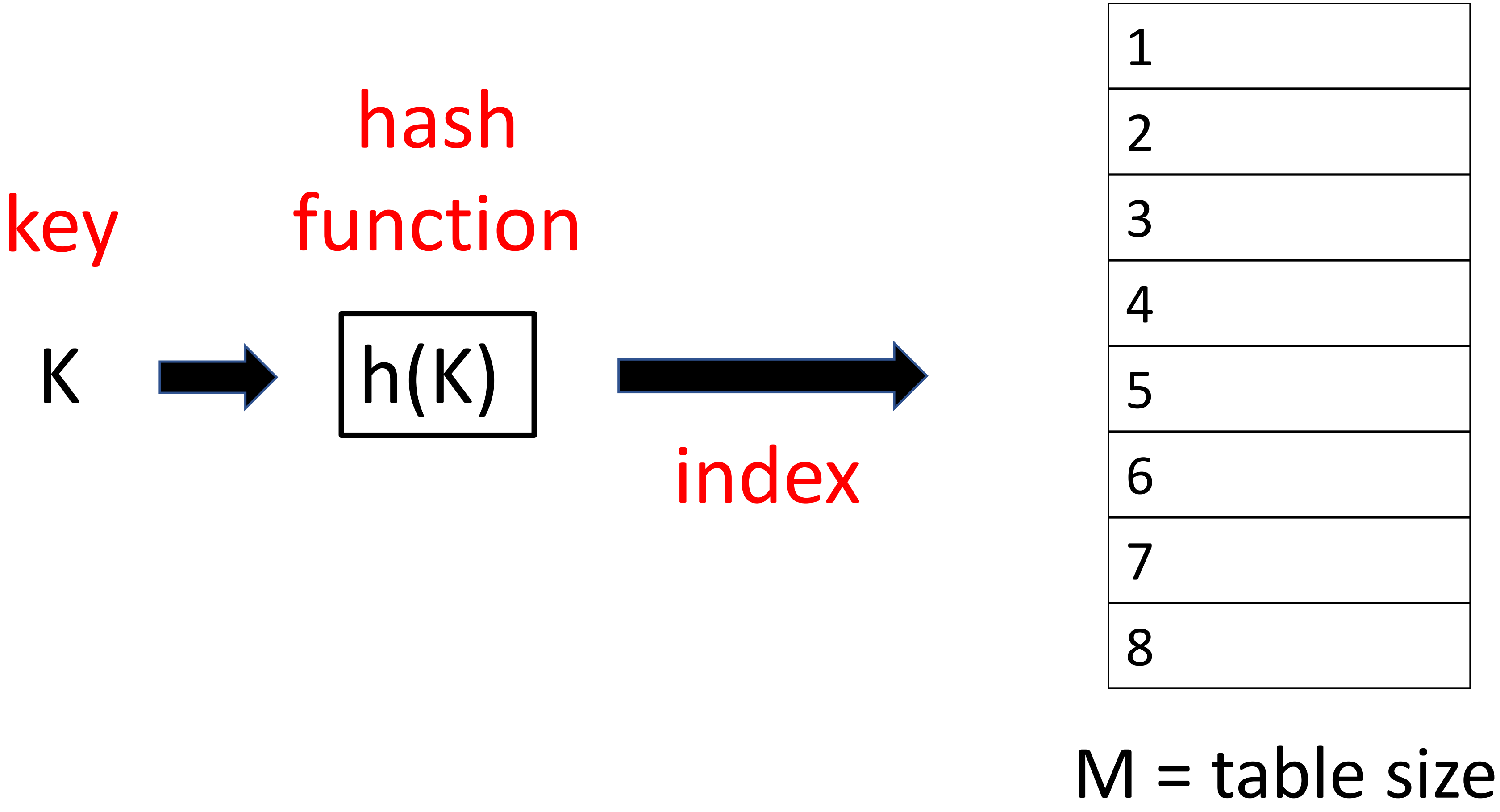
# Hash Tables

- A hash function performs the mapping of a key to a storage location in an array
- The simplest hash function converts each letter in a key’s string to an int, and then performs a mod operation on the sum
  - We don’t know how many keys we’ll see, but we have a limited amount of storage for all of our data.
  - The mod function ensures we stay within the bounds of our storage array

# Hash Tables

- Hash Functions
  - Division Method (what we've done)
    - h(K) = K mod M
    - Choose M as a prime (not near a power of 2)
  - Multiplication Method
    - h(K) = floor(M * frac(K * 0.6180339887))
    - Choose M = 2^m

# Hash Tables



# Hash Tables

- Collisions happen when two different keys map to the same index of the hash table
  - We are guaranteed collisions when the number of keys exceed the number of hash table entries
  - input (N) > table size (M)
  - Collision: h(k1) = h(k2) where k1 != k2

# Hash Tables

- Birthday Paradox
  - Given 23 random people's birthdays, we have a 50% chance of two of those being the same
  - Given 23 random keys in a set of N ~= 365, we have a 50% chance of a collision

# Hash Tables

- Assume we have a **perfect hash** function
  - All keys entered into this hash function will be mapped without collision or wasted space
  - ex) Store 100 sequential phone numbers in a table of size 100
- An **imperfect hash** function fails on one of two fronts
  - k1 != k2, where h(k1) = h(k2)
  - N > M or M > N
- The only way to remove collisions is to allow the hash table to grow arbitrarily large
- So let's think of better ways to live *with* them

# Hash Tables

Handling Collisions

- Open Hashing (wrapping)
  - Using a secondary linear probing function, resolve collisions by finding the next empty index to insert to
  - Probe wrap-around should allow index n-1 to be adjacent to index 0
  - Probing function necessary for insert, search
- Closed Hashing (chaining)
  - Use a secondary data structure to store collisions
  - A linked list is a standard implementation, but any other data structure may feasibly be used

# Hash Tables

- Every hash table has a **load factor** = N/M = α
- Open Hashing
  - Average number of probes for a **hit**
    - $\frac{1}{2}(1 + \frac{1}{1-\alpha})$
  - Average number of probes for a **miss**
    - $\frac{1}{2}\left(1 + \frac{1}{(1-\alpha)^2}\right)$

| Load factor α | ½ | 2/3 | ¾ | 9/10 |
|---|---|---|---|---|
| Search hit | 1.5 | 2 | 3 | 5.5 |
| Search miss | 2.5 | 5 | 8.5 | 55.5 |

# Hash Tables: Complexity

- Average performance for inserting and searching hash tables is O(1)
  - Constant time access!
- Worst case performance is O(n)
  - found when all keys hash to the same index, where a O(n) search along a linked list may be performed.
  - found when probing through all indices of the hash table before finding the desired key

# Hash Tables

- How big should a hash table be?
- Firstly, chose how you handle collisions:
  - Open Addressing (wrapping)
    - For N inputs, you would require at least N indices
  - Chaining
    - For N inputs, decide how many linked list traversals you can live with
    - M should be the nearest prime to N divided by the # of linked list nodes traversed

# Last Lecture Topic Ideas

- Heaps (deeper dive)
- Priority Queues
- Dictionaries Using Hash Tables
- Command Line Usage
- Regular Expressions
- Name Spaces
- Debuggers (GDB, Valgrind)
- suggestions?...