# Hashing

## Constant-Time — $\Theta(1)$ — Searching

CSCI 3700 — Data Structures and Objects

Department of Computer Science and Information Systems
Youngstown State University

Robert W. Kramer

# Outline

# What We Have Seen So Far

## Summary of searching algorithms

- Sequential searches
  - Sequential, sentinel, probability, ordered sequential
  - $O(n)$ time
- Binary searches
  - Regular and forgetful
  - $O(\lg n)$ time
  - List must be sorted

# Some Other Methods

## Other search algorithms you can use

- Interpolation search
  - Similar to binary search
  - Estimates where key should be within bounds
  - $O(\lg \lg n)$ to search on average, $O(n)$ in worst case
  - List must be sorted
- Trie search
  - String-based search
  - Uses a tree of lists
  - $O(|s|)$ time
  - Generally not space efficient

# What Information Do We Have?

- Sequential search
  - This is / this is not the key
- Binary / algorithmic search
  - Key is earlier / later in the list

- Relative key information
  - Best worst-case scenario: $O(\lg n)$

What about absolute information?

# Pinpointing Location

## Using absolute location information

- Consider questions around the home:
  - Where is the peanut butter?
  - Where are the Band-Aids?
  - Where is the TV remote?
- Search is often limited
  - Cupboard, pantry, shelf, freezer, etc.
  - Items usually organized

# Hashing

- Hashing provides absolute location
  - Search in one specific part of the list
- Key is converted to number
  - Number indicates location in list
- Can be very efficient
  - Can have $O(1)$ search on average

# Issues With Hashing

## Must be able to answer these

- Can we always convert keys to numbers?
  - Many ways to do so
  - Some are better than others
- What if two keys are converted to the same number?
  - Need to handle *collision resolution*

# Mathematics Of Hashing

- Analysis of hashing involves math
  - Probability, calculus, basic number theory
- Not covered here
  - See supplementary material if interested
- End result of analysis
  - All constants (table size, multipliers, etc.) should be *prime*
  - Two properties must be met

# Hashing Properties

## You want your hash table to have these properties!

- Hashing function must distribute keys evenly
  - Gaps are also evenly distributed
- Table must not exceed $x\%$ full
  - You can choose the value for $x$
  - Guarantees gaps are not too far apart

**If these are met...**

Search, insert and remove are all $O(1)$

# Hashing Necessities

- Hash table
  - Fixed-size list of items, or
  - Fixed-size list of item lists
- Hash function
  - Converts keys to locations in hash table
- Collision resolution
  - Handling multiple keys in same location

# Hash Functions

## Turning your key into hash

- Any function that returns an (unsigned) integer
  - Some work better than others!
- Categories of hash functions
  - Simple-to-compute
    - One line of code, basic math
  - Less-simple-to-compute
  - String-to-number conversions

# Some Basic Hash Functions

## Infinitely many functions, these eight work well in practice

- Simple-to-compute
  - Direct and subtraction
  - Mod division
  - Pseudorandom number generation
- Less-simple-to-compute
  - Digit extraction
  - Midsquare
  - Folding
  - Rotation

# Direct and Subtraction Hashing

## The simplest hash functions

- Direct hashing
  - $p = key$
  - Example: calendar
    - Event date is key
- Subtraction hashing
  - $p = key - c$, $c$ is a constant
  - Example: checkbook
    - Subtract 101 from check number

# Mod Division

## Works well alone or with other techniques!

- $p = key \bmod c$, $c$ is a constant
- $c$ is often the table size
- Range of remainder values
  - $0 \ldots (c-1)$
  - Same range as valid array positions!
- Often used with other methods
  - Maps result onto table

# Pseudorandom Number Generation

This is what `rand()` does

- $p = (key \cdot a + c) \bmod t$
  - $t$ is table size
  - $a$ and $c$ should be fairly large
  - $a$, $c$, $m$ should be *relatively prime*
- Distribution appears random
- Used for most random number generators

# Digit Extraction

## The first of the less-simple techniques

- Let $s(key)$ select and concatenate certain digits from *key*
  - Combination of several $n/10^a$, $n \cdot 10^b$ and $n \bmod 10$ actions
- Digits should have good variance
- Should have $s(n) \gg t$, $t$ is table size
- Compute $p = s(key) \bmod t$

# Midsquare

## No, not the center of a regular quadrilateral

- Let $s_4(key)$ select four digits from *key*
    - More than four causes overflow
- Square the number
- Pad the result (add zeroes on left) to make an 8-digit number
- Select middle four digits
    - These are mixture of original four digits
- Use mod division to map result to table size
- Result: $p = \left(\lfloor s_4(key)^2/100 \rfloor \bmod 10000\right) \bmod t$

# Folding

## Like origami, with numbers

- Split *key* into equal-size chunks
  - Zero-pad if necessary
- Add chunks
- Use mod division to map to table size

- Shift folding
  - $123456 \rightarrow 12 + 34 + 56 = 102$
- Boundary folding
  - $123456 \rightarrow 12 + 43 + 56 = 111$

# Rotation

This subtitle omitted due to insufficiently humorous joke

- Split *key* into two parts
  - Parts do not need to be equal
- Swap parts
  - Net effect: digits are rotated
  - Uses $n \bmod 10^a$, $n/10^a$ and $n \cdot 10^b$
- Use mod division to map to table size

# String Hashing

## Converting strings to numbers

- Convert string into number sequence
  - ASCII, Latin-1, Unicode encodings
- Use some form of folding
  - Partial fold
  - Weighted fold
- Don't just add the numbers!

# Collision Resolution

## Keys, cars, fermions. . .

- Multiple keys may be mapped to same position
  - Keys cannot occupy same position simultaneously*
- Two main classes of collision resolution
  - Rehash to find new location
  - Extend location to accommodate

# Clustering

Before we start talking about collision resolution techniques. . .

- Primary clustering
  - Many keys map to same location
  - Indicative of bad hash function
- Secondary clustering
  - Colliding keys interfere with nearby locations

# Finding a New Location

"I'm looking for a new direction" — Echo and the Bunnymen

Four common techniques

- Linear probe
- Quadratic probe
- Pseudorandom probe
- Key offset

# Linear Probe

## This looks familiar...

- Use hash function to get initial location $p$
- Search locations $p$, $p + 1$, $p + 2$ etc.
- Wrap around end of table
- This is sequential search!

# Quadratic Probe

## This also looks familiar...

- Use hash function to get initial location $p$
- Search locations $p$, $p + 1$, $p + 4$, $p + 9$ etc.
- Wrap around end of table
- This is also sequential search!
- Tries to prevent secondary clustering

# Pseudorandom Probe

## Another attempt to reduce secondary clustering

- Begin with initial position $p$
- Carry out pseudorandom calculation
  - $p_{new} = (p_{old} \cdot a + c) \bmod t$
- Repeat until *key* is found

# Collision Path

## Where do we search for open locations?

- Starting at position $p \ldots$
  - What sequence of locations is checked?
- Linear, quadratic, pseudorandom probing
  - All keys colliding at position $p$ search the same sequence of locations

**Note**

$n$ colliding keys $\rightarrow O\left(n^2\right)$ primary collisions

# Key Offset

## An attempt to minimize the length of the collision path

Collision? Use a second hash function

- Add result of second hash to $p$
  - Result cannot be multiple of table size!
- Colliding keys should map to different locations
- Use other probe techniques if key offset fails to resolve

# Buckets

## One position, fixed number of slots

Allow each location to store $b > 1$ items

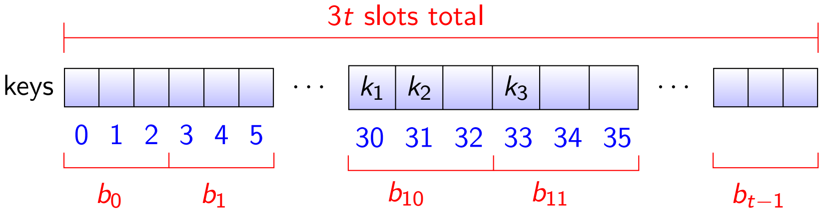
- Locations are *buckets*
- Multiply table size by $b$
- Feed key to hash function
- Multiply result by $b$
- Collision? Linear probe!

This reduces secondary clustering

# Bucket Example

## This slide took over two hours to prepare

An example with $b = 3$:



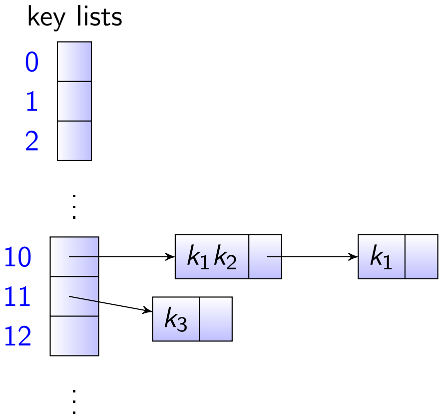

$k_1$ hashes to position 10
$k_2$ hashes to position 10
$k_3$ hashes to position 11 (no secondary collision)

# Linked Lists

## Like buckets but better

- Each location is a list of items
- Lists grow or shrink as needed
- No secondary collisions!
- Lists searched sequentially
- Redefine "x% full"
  - $keyCount \leq y \cdot tableSize$

# Linked List Example



# Summary

- Hashing can provide constant-time search
- Keys must be evenly distributed
- Table cannot get too full
- Table size should be prime
- Need hash function to convert key to position
- Need method for handling collisions