# Hash Tables
# Open Addressing

# HashTables

- A hash table is a form of a map that has better time complexity
- A hash table consists of
  - an array of size $N$
  - an associated hash function $h$ that maps keys to integers in [0, N-1]
  - A “collision” handling scheme
- Hash Function
  - $h(x) = x\%N$ is such a function for integers
  - item $(k, v)$ is stored at index $h(k)$
- Collision Handling
  - A “collision” occurs when two **different** keys hash to the same value

# Separate Chaining

- Idea: each spot in hashtable holds a map of key value pairs when the key maps to that hashvalue.
- Replace the item if the key is the same
- Otherwise, add to map
- Generally do not want more than about number of objects as size of table
- Chains can get long

# Open Addressing
# Probing

- Store only <K,V> at each location in array
  - No awkward lists
- If key is different and location is in use then go to a different location in array
    - What different location?
      - Repeat until free location found
- If you stored <K,V> in different location, how do you find it?

# Probe distance

- When location is in use need a formulaic way to find a new location
  - Linear Probing
    - Simple but has problems
  - Quadratic Probing
    - Not as simple, fewer problems
  - Double Hashing
    - Requires two hash functions, best

# Linear Probing

- Compute hash location for Key
- Let loc=h(key), q=0
  - q a.k.a probeCount
- Repeat:
  - if (loc+q)%N unoccupied, put in Pair .. Done
  - if key is same, replace value .. Done
  - q++; // Next spot

# Linear Probing Practice

- Put the following data into a hashtable using linear probing
  - Hashtable size = 17
  - h(x) = x % 17
    - What is the worst case for number of probes?

| | | | |
|---|---|---|---|
| <4,A> | <13,B> | <39,C> | <32,D> |
| <21,E> | <40,G> | <31,H> | <30, J> |
| <14,K> | <3,L> | <48,M> | <20,N> |

# Linear ==> Quadratic

- Linear probing suffers from "Primary clustering"
  - the bigger the cluster gets, the faster it grows
- So idea, rather than place=(loc+q) make place=loc+q*q
  - Logic -- take bigger and bigger hops to escape from primary cluster
  - "Quadratic probing"

# Quadratic Probing

- Compute hash location for key
- let loc=h(key), q=0
- Repeat:
  - if (loc+q*q) unoccupied, put in Pair .. Done
  - if key is same, replace value .. Done
  - q++

# Quadratic Probing Example

- Suppose
  - hashtable size is 7
  - h(t)=t%7
  - add:
    - <3,A>
    - <10,B>
    - <17,C>
    - <24,Z>
    - <3,D>
    - <4,E>

# Quadratic ==> Double Hashing

- Clustering still happens, just not as bad
  - "secondary clustering"
    - because every entry uses the same jumping sequence
- So need to get different jump sequences.
  - define a new hashing function h2 that gives the jump sequence for a key
  - Suppose two keys k1, and k2 such that h1(k1)=h1(k2)
    - Then probably h2(k1)!=h2(k2) so the jump sequences are different
    - Hence, avoid primary and secondary clustering

# Double Hash Probing

- Define a second hashing function h2(key)
  - h2 is in range P...Q
    - P > 0, usually P > 1, but 1 is OK
    - Q > P, usually Q < N, Q>N ok, just annoying
- Let `q=0; loc=h1(key); inc=h2(key)`
- `Repeat:`
  - if `loc+q*inc` unoccupied, put in Pair .. Done
  - if key is same, replace value .. Done
  - `q++`

# Double Hash Practice

- Put the following data into a hashtable using double hash probing
  - Hashtable size = 17
  - h(x) = x % 17
  - h2(x) = (x%20)+2
    - What is the worst case for number of probes?

| | | | |
|---|---|---|---|
| <4,A> | <13,B> | <39,C> | <32,D> |
| <21,E> | <40,G> | <31,H> | <30, J> |
| <14,K> | <3,L> | <48,M> | <20,N> |

# Probing Distance (Summary)

- Given a hash value $h(x)$, linear probing generates $h(x),\ h(x)+1, h(x)+2, \ldots$
  - Primary clustering – the bigger the cluster gets, the faster it grows
- Quadratic probing – $h(x),\ h(x)+1, h(x)+4,\ h(x)+9, \ldots$
  - Quadratic probing leads to secondary clustering, more subtle, not as dramatic, but still systematic
- Double hashing
  - has neither primary nor secondary clustering
  - But you need two hashing functions
    - each hash takes some time
      - if using Horner's, then for second function just change the multiplier and change the modulus (and add one)
      - hash function in Java

# Performance Analysis for probing

- In the worst case, searches, insertions and removals take $O(n)$ time
  - when all the keys collide
- The load factor $\alpha$ affects the performance of a hash table
  - expected number of probes for an insertion with open addressing is $\frac{1}{1-\alpha}$
- Expected time of all operations is $O(1)$ provided $\alpha$ is not close to 1
  - NOTE: cheating here O() is about true worst case

# Removing Items

- In separate chaining just remove.
- Probing: cannot simply delete as positions are dependent on what was there are time inserted
- So rather than set position empty on delete, replace item with "tombstone"

# Probing vs Chaining

- Probing is significantly faster in practice
  - Why? locality of references
    - much faster to access a series of elements in an array than to follow the same number of pointers in a list
- Efficient probing requires tombstoning
  - de-tombstoning??
    - like defragmenting a hard disk