CS C341 / IS C361
Data Structures & Algorithms

Dictionary Data Structures - Hashing

Hash Tables

- Bucketing and Hashing
- Separate Chaining
- Hash functions
  - Good hash functions
  - Universal hash functions

DICTIONARY DATA STRUCTURES

- Consider an un-ordered dictionary:
  - Typically, elements (and keys) are unique.
  - Need to optimize find, add, and delete operations (typically in that order).
- Simplest case:
  - The universe of keys, U, is a range of integers: [lo .. hi]
  - Representation:
    - In this case, a table T indexed from 0 to |U|-1 is a good representation.
      - T[k-lo] contains element with key k.

Hash Tables

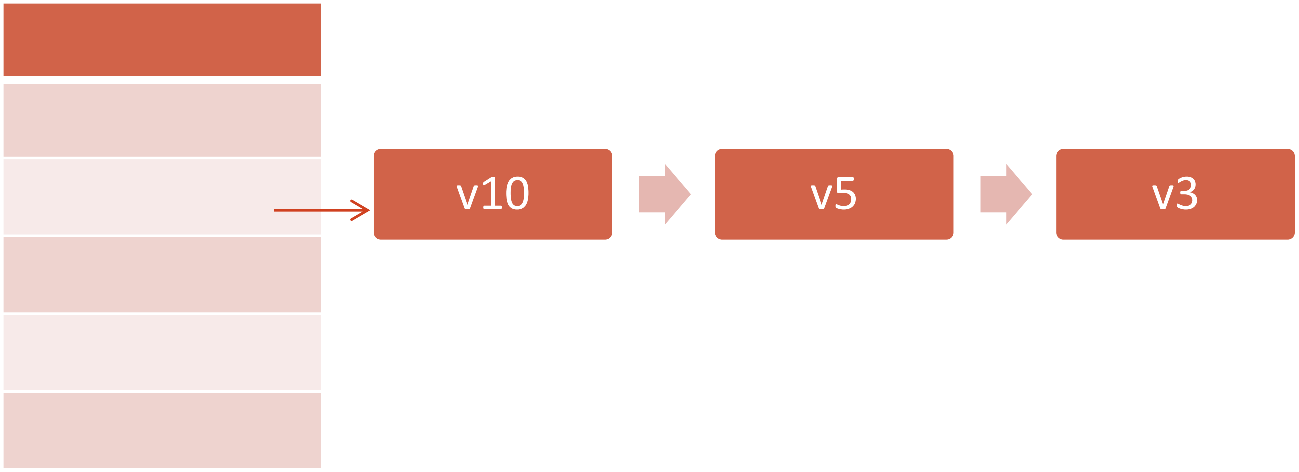
- Suppose the set of keys stored in a Dictionary is small compared to the universe (range) of keys:
  - Need a good mapping from (a large set of) values to (a small set of) integers – i.e. indices
  - Typically referred to as a *hash function*
    - *h : U --> { 0, 1, …, m-1 }*

    where m is the size of the table.

  If h is 1-to-1, then each bucket is associated with a unique key
- Collision:
  - Two keys may hash to the same slot (if h is not 1-to-1)
- Goal:
  - h must be uniformly random to minimize collisions – why?
- Collision Resolution?

Hash Tables

- Collision Resolution by Chaining
  - Representation: Table of linked lists.
  - All elements that hash to a (given) slot are chained in an un-ordered linked list.



Hash Tables

- Implementation:
  - Initialize a table (i.e. array) T of empty (linked) lists.
  - Element find(Key k, Hashtable T)
    - return linearSearch(k, T[h(k)]);
  - void add(Element e, Hashtable T) // e.k is key
    - insertInLinkedList(e, T[h(e.k)]);
      - If *add* also acts as update (i.e. overwrite) then insertion may traverse the entire list (in the worst case).
      - Else insertion may be at the beginning (i.e. ignore duplicates)
  - void delete(Key k, Hashtable T)
    - deletefromLinkedList(k, T[h(k)]);

Hashing – Chaining - Analysis

- Load Factor
  - Given a table T with m slots, if n elements are stored define Load factor $\alpha = n/m$
- Assumption:
  - h(k) can be computed in O(1) time
- Worst case time for find (or add or delete):
  - O(n)
    - Why?
- Average case time:
  - ?

Hashing – Chaining - Analysis

- Assumption:
  - Any given element is equally likely to hash into any of the m slots (independent of other elements)
    - Simple uniform hashing (SUH)
- Average case time for *unsuccessful find*: $\Theta(1+\alpha)$
  - Under SUH, average time for *unsuccessful find* for a key is
    - Average time to search to the end of one of the m lists
      - Average length of such a list is α
    - Thus the total time is: $\Theta(1+\alpha)$

Hashing – Chaining - Analysis

- Average case time for *successful find*: $\Theta(1+\alpha)$
  - Assume add puts element at the end of list
    - Needed if add overwrites existing element - if any.
  - Expected number of elements examined
    - 1 + Expected number of elements examined during add
      - $1 + (i-1)/m$ for the i^{th} element
    - Do the average over n elements
- Average case time for *successful find*: $\Theta(1)$
  - if we allow multiple elements with same key
    - Why?

WHAT IS A GOOD HASH FUNCTION?

- Example hash function 1 (for natural numbers):
  - h(k) = k mod m
  - When is this a good hash function?
    - Counter-examples:
      - $m = 2^p$ for some p
      - $m = 10^p$ for some p when keys are decimal numbers
  - Good choice for m
    - A prime number that is not close to a power of 2.
      - Why?
- Example hash function 2 (for natural numbers):
  - h(k) = floor(m*(k*A mod 1))
    - where A is a constant and 0<A<1
  - Don Knuth's recommendation:
    - $A = (\sqrt{5} - 1)/2$

What is a Good Hash Function? [2]

- Symbol tables (in compilers/assemblers) often are implemented as hash tables:
  - Records of Symbols and their attributes (e.g. type, scope, address etc.)
- What is a good hash function for strings?
  - E.g. Sum of ASCII values of characters in a string modulo m, where m is the size of the table.
    - Is this good? Why or Why not?
  - E.g. $(x_0 * a^{k-1} + x_1 * a^{k-2} + \ldots + x_{k-2} * a + x_{k-1})$ mod m given string $(x_0, x_1, \ldots, x_{k-1})$ and chosen constant a
    - Can you implement this in O(k) time ?
    - What should be a good choice of a?

WORST CASE KEY SETS

- It is possible that given a hash function all or most keys hash to the same slot resulting in $\Theta(n)$ access time for each key.
  - E.g. all variables in a program have similar "characteristics"
- Any fixed hash function may exhibit worst case behavior for a set of keys
  - Consider an adversary who – knowing the given hash function – chooses a set of keys such that they all hash into the same slot.
- One way to improve the latter situation:
  - Choose the hash function randomly at run time in a way that is independent of keys
  - This is referred to as universal hashing

Universal Hashing

- A solution to this approach:
  - Choose a hash function randomly and independently from the set of keys
    - i.e. every hash table instance chooses its hash function – say at the time of creation or initialization –
    - And the choice is made uniformly randomly from a class of hash functions.
  - Consider the adversary again:
    - Adversary doesn't know the hash function – so cannot bias the input (of course, incidental bias can still happen!)

UNIVERSAL HASH FUNCTIONS

- Let H be a finite collection of hash functions that map a given universe U of keys into the range { 0,1,…,m-1}.
  - This collection is said to be a universal family,
    - if for each pair of distinct keys x and y in U, and a hash function h chosen uniformly randomly from H
    - Probability (h(x)=h(y)) <= 1/m
- The family H (defined below) is universal.
  - $H = \{ h_{a,b} \mid 0 < a < p \text{ and } 0 <= b < p \}$
    - where $h_{a,b}(k) = ((a*k+b) \bmod p) \bmod m$
    - and p is a prime such that $|U| <= p < 2*|U|$
- (Proof omitted)

CS C341 / IS C361
Data Structures & Algorithms

Dictionary Data Structures - Hashing

Open Addressing

- Analysis
  - Probing
  - Unsuccessful Find
  - Successful Find

Bloom Filters

- Motivation
- Implementation
- Analysis
- Applications.

TERMINOLOGY

- The technique of chaining elements that hash into the same slot is referred to by different names:
  - Separate Chaining
    - for obvious reasons
  - Open Hashing
    - because number of elements is not limited by table size
  - Closed Address Hashing
    - because the location of the bucket (i.e. the address) of an element is fixed
      - Consider an element e that is added, removed, and added again:
        - it will get added to the same bucket.

Open Addressing (a.k.a. closed hashing)

- Fixed Space
  - Fixed Table size and
  - each table location can contain only one element
- Addressing by Hashing
  - Same as in Separate Chaining
- Probing (for a vacant location) in case of collision

OPEN ADDRESSING (A.K.A. CLOSED HASHING)

- add(Element e, Hashtable T)   // Generic procedure

```
// e.key is key; h is hash function
a = h(e.k);
if T[a] is empty then { T[a]=e; return; }
j=0;
repeat  {
   j++;
  b = getNextAddr(a,k,j);
} until (T[b] is empty)      // Will this terminate?
T[b] = e;
```

OPEN ADDRESSING – PROBING SCHEMES

// m denotes table size; typically m is chosen to be prime

- Linear Probing:

  getNextAddr(a,k, j) { return (a+j) mod m; }

- Quadratic Probing

  getNextAddr(a,k,j) { return (a+j^2) mod m; }

- Exponential Probing

  getNextAddr(a,k,j) { return (a+2^j) mod m; }

- Double Hashing

  getNextAddr(a,k,j) { return a+j*h_2(k) mod m; }

  // h_2(k) is the secondary hash function

  // h_2(k) must be non-zero

  // e.g. $h_2(k) = q - (k \bmod q)$ for some prime $q < m$

Open Addressing

- Implementation Caveat:
  - add as defined may not terminate!
    - Must check whether all ***m*** locations have been probed
      - Could be expensive!
    - Alternatively, may use a count of non-empty locations.
      - Will work only if the probing sequence covers all locations

Exercise: Handle termination: use simple heuristics(s). **End of Exercise.**

Open Addressing

- Define find.
  - Similar to add:
    - hash
    - if element found return it;
    - if empty return INVALID;
    - otherwise probe until element found or empty slot.
      - return accordingly.
  - Termination?

Open Addressing

- How is deletion done?
  - Deleted slots must be marked *deleted*
    - ***deleted*** flag different from ***empty*** flag for probing procedure to work
      - **find** will treat ***deleted*** slots as ***empty*** slots
  - This won't allow re-use of ***deleted*** slots
    - How do you recover deleted slots?
      - **add** can be modified to fill in any *deleted* slot encountered in a probing sequence
        - This may not cover all deleted slots
      - **delete** can be implemented such that subsequent entries in a probing sequence are pulled in.
- How does your deletion scheme affect further probes?

Open Addressing – Analysis of Probing

- Probing sequence:
  - Sequence of slots generated: S[k,0], S[k,1],..S[k,m]
- Probing requirements:
  - Utilization:
    - The probing sequence must be a permutation of 0,1,…m-1
  - Uniform hashing assumption:
    - Requires that each key may result in any of the m! probing sequences

Open Addressing – Analysis of Probing [2]

- Linear Probing
  - Slot for the j^{th} probe in a table of size m

    $S[k,j] = (h(k) + j) \bmod m$
  - Long runs of occupied slots build up
    - If an empty slot is preceded by j full slots,
    - then the probability this slot is the next one filled is $(j+1)/m$
    - instead of $1/m$ (in a table of size m)
  - Effect known as (Primary) clustering
- This is not a good approximation of uniform hashing

Open Addressing – Analysis of Probing [3]

- Quadratic Probing
  - Slot for the j^{th} probe in a table of size m

    $S[k,j] = (h(k) + j^2) \bmod m$
  - Clustering effect milder than Linear probing
    - Effect known as secondary clustering
  - But the sequence of slots probed is still dependent on the initial slot (decided by the key)
    - i.e only m distinct sequences are explored
- Generalize:
    - $S[k,j] = (h(k) + a*j + b*j^2) \bmod m$

Exercise: Can you choose a, b, and m such that all slots are utilized?

Exercise: Repeat (very similar) analysis for Exponential Probing.

OPEN ADDRESSING – ANALYSIS OF PROBING [4]

- Double Hashing
  - Slot for the j^{th} probe in a table of size m

    $S[k,j] = (h_1(k) + j*h_2(k)) \bmod m$
  - Probing sequence depends on k in two ways
    - So, probing sequence depends not only on initial slot

      i.e. m*m probing sequences can be used.

    This results in behavior closer to uniform hashing
  - If $\gcd(h_2(k),m) = d$ for some key k,
    - then the sequence will explore only (1/d)*m slots
      - Why?
    - So, choose (for instance):
      - m as a prime, and ensure $h_2(k)$ is always < m
- Can you extend this to a sequence of hashes $h_1(k)$, $h_2(k)$, $h_3(k)$, … ?

Open Addressing – Analysis - Unsuccessful Find

- Given: open-address table with load factor $\alpha = n/m < 1$
- Assumption: Uniform Hashing
- Expected number of probes in an unsuccessful find is at most

  $1/(1-\alpha)$
- Proof:
  - Last probed slot is empty; all previous probed slots are non-empty but do not contain the given key
  - Define p_j as the probability that exactly j probes access non-empty slots
    - Then the expected number of probes is $1 + \sum_0^\infty j*p_j$
  - If q is defined as the probability that at least j probes access non-empty slots then $\sum_0^\infty j*p_j = \sum_1^\infty q_j$

2/22/2014

Open Addressing – Unsuccessful Find

- Proof: (contd.)
  - The expected number of probes is

    $1 + \sum_0^\infty \; j*p_j = 1 + \sum_1^\infty q_j$
  - With uniform hashing

    $q_j = (n/m) * ((n-1)/(m-1)) * \ldots ((n-j+1)/(m-j+1))$

    $<= (n/m)^j$
  - Then the expected number of probes is

    $1 + \sum_1^\infty q_j <= 1 + \alpha + \alpha^2 + \alpha^3 + \ldots$

    $= 1 / (1 - \alpha)$

Open Addressing – Analysis - successful Find

- Given: open-address table with load factor $\alpha = n/m < 1$
- Assumptions: Uniform Hashing; All keys are equally likely to be searched
- Expected number of probes in a successful find is at most $1/\alpha + (1/\alpha)*\ln(1/(1-\alpha))$
- Proof:
  - Simplifying assumption :
    - A successful find follows the same probe sequence as when the element was inserted
    - When is the assumption reasonable?
  - If k was the $(j+1)^{st}$ key to be inserted
    - then the expected number of probes in finding k is given by the previous theorem (on unsuccessful find)

$$1/(1 - (j/m)) = m/(m-j)$$

OPEN ADDRESSING – ANALYSIS - SUCCESSFUL FIND

- Proof: (contd.)
  - Expected number of probes in finding the key that was inserted as the $(j+1)^{st}$ is $m/(m-j)$
  - Average over all n keys in the table

$$(1/n) \sum_{j=0}^{n-1} (m/(m-j)) = (m/n) * \left(\sum_{j=0}^{n-1} (1/(m-j))\right)$$
$$= (1/\alpha) * (H_m - H_{m-n})$$

where H_m is the mth Harmonic number.

  - Since $\ln(j) \le H_j \le 1 + \ln(j)$

$$(1/\alpha) * (H_m - H_{m-n}) \le (1/\alpha) * (1 + \ln m - \ln(m-n))$$
$$= (1/\alpha) + (1/\alpha) * \ln (m/m-n)$$
$$= 1/\alpha + (1/\alpha)*\ln(1/(1-\alpha))$$

Re-Hashing

- Hash tables support efficient find operations:
  - Average case time complexity is O(1) if load factor is low
    - Load factor must be < 1 for separate chaining
- In practice,
  - Load factor must be < 0.75 to expect good performance.
- What if the hash table is nearly “full”?
  - Extend the hash table (i.e. increase its size)
    - Can the new hash function assign the old values to the same buckets as before?
      - bucket addresses must change for a good distribution?
  - Re-insert all the elements in the table
    - Referred to as ***re-hashing.***

RE-HASHING

- Cost of Rehashing
  - O(max(m,n)) time – typically O(n) as table is nearly full.
    - Amortized Cost: O(1) time per element
  - But response time at the point of rehashing is bad:
    - allocation and copying of all the values takes O(n) time between two operations.
    - Or between the request for an operation and the response.
  - This is bad for applications requiring
    - bounded (worst case) response time
- What should be the size of the extended table?
  - Typical choice: 2*|T|
  - Trade-offs: ???

Bloom Filters - Motivation

- Tradeoff: Space vs. (In)Correctness
  - i.e. storage space for the table vs. false positives (membership)
- Example Problem: Stemming of words in search engine indexing:
  - e..g. plurals stemmed to singular; all parts of speech stemmed to one form
    - 90% of cases can be handled by simple rules
      - Rest - the exceptions – need a dictionary lookup
    - Suppose dictionary is large and must be stored in disk

Bloom Filters - Motivation

- Consider this outline for stemming :

  for each word w

      if (w is an exception word)

    then   getStem(w,D)

    else apply-simple-rule(w)

  Need dictionary lookup on disk

- Cost for checking exceptions:
  - N * Td   where
    - N is # words and
    - Td is lookup time (on disk)

Bloom Filters - Motivation

- Suppose we can trade-off space for false positives (in lookup):

```
for each word w
    if (w is in Dm )          // in-memory lookup (probabilistic)
    then { s = getStem(w, Dd);    // disk lookup (deterministic)
      if invalid(s) then apply-simple-rule(w);
    } else { apply-simple-rule(w); }
```

- Cost for checking exceptions:
  - N * Tm + (r + f)*N*Td
    - r is the proportion of exception words
    - f is false positive rate
    - Tm is lookup time in memory
    - Td is lookup time on disk
  - Time Saved: (1 – r –f) * (Td – Tm) / Td

Bloom Filters – An implementation

- Hash table is an array of bits indexed from 0 to m-1.
  - Initialize all bits to 0.
  - insert(k):
    - Compute $h_1(k), h_2(k), \ldots, h_d(k)$ where each h_i is a hash function resulting in one of the m addresses.
    - Set all those addressed locations to 1.
  - find(k):
    - Compute $h_1(k), h_2(k), \ldots, h_d(k)$
    - If all addressed locations are 1 then k is **found**

      Else k is **not found**

Not necessarily correct!

Always correct.

BLOOM FILTERS - ANALYSIS

- Consider a table H of size m.
- Assume we use d "good" hash functions.
- After n elements have been inserted, the probability that *a specific location is 0* is given by
  - $p = (1 - 1/m)^{dn} \approx e^{-dn/m}$ // Why?
- Let q be the proportion of 0 bits after insertion of n elements
  - Then the expected value E(q) = p
- Claim (w/o proof):
  - With high probability q is close to its mean.
- So, the false positive rate is:
  - $f = (1-q)^d = (1-p)^d = (1 - e^{-dn/m})^d$

Bloom Filters

- The data structure is probabilistic:
  - If a value is not found then it is definitely not a member
  - If a value is found then it may or may not be a member.
- The error probability can be traded for space.
  - In practice, one can get low error probability with a (small) constant number of bits per element: (1 in our example implementation) .
- Applications:
  - Dictionaries (for spell-checkers, passwords, etc.)
  - Distributed Databases – exchange Bloom Filters instead of full lists.
  - Network Processing – Caches – exchange Bloom Filters instead of cache contents
  - Distributed Systems – P2P hash tables : instead of keeping track of all objects in other nodes, keep a Bloom filter for each node.

LAS VEGAS VS. MONTE CARLO

- Quicksort:
  - Randomization for improved performance – correctness not altered
- Hashtables (for unordered dictionaries) :
  - Any 1-to-1 mapping will yield a table but a good hash function should yield a "uniformly random" distribution
  - Universal hashing chooses hash function "randomly"
- Both of the above are optimizations:
  - Such techniques are referred to as Las Vegas techniques.
- Monte Carlo Technique
  - Bloom Filter - Randomization yields a probabilistic algorithm that does not always produce correct results.