COSC160: Data Structures
Hashing Structures

Jeremy Bolton, PhD
Assistant Teaching Professor

GEORGETOWN UNIVERSITY

Outline

I. Hashing Structures
   I. Motivation and Review
II. Hash Functions
III. HashTables
   I. Implementations
   II. Time Complexity
IV. Collisions
   I. Resolution Schemes

GEORGETOWN UNIVERSITY

Retrieval Time Review

- Unordered Lists
- Trees or ordered lists
- Can we improve?

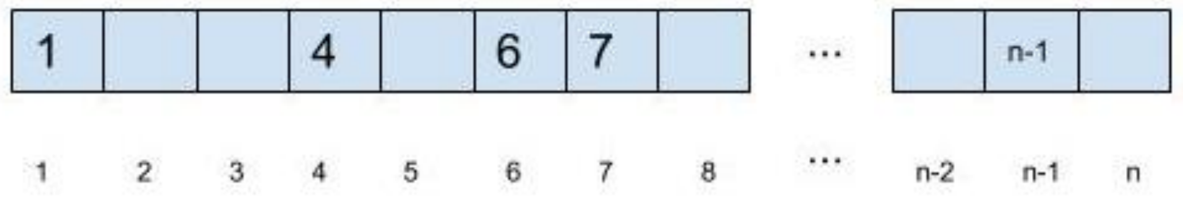
Motivation: Simple Example

- Suppose we wanted to store a set of unique numbers within the range 1 – 1,000
- Is there a structure and storage scheme that would permit searching, inserting and removing in O(1) time?
  - Hint: the answer is yes! Motivation: Constant Time Example

GEORGETOWN UNIVERSITY

Motivation: Simple Example

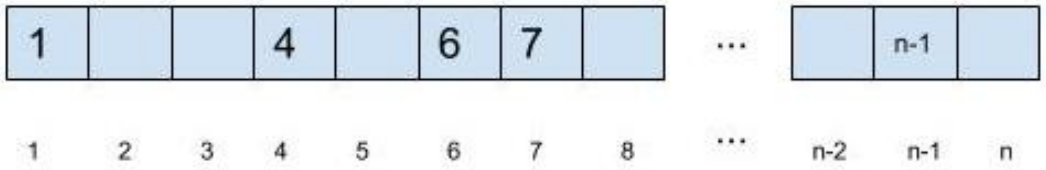
- Simply use an array with indices 1 – 1000.



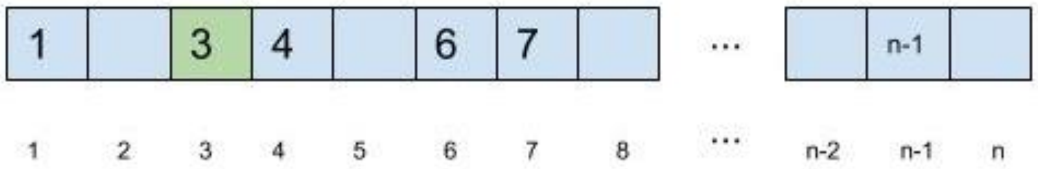
GEORGETOWN UNIVERSITY

Motivation: Simple Example

- Insertion



- Example: insert 3

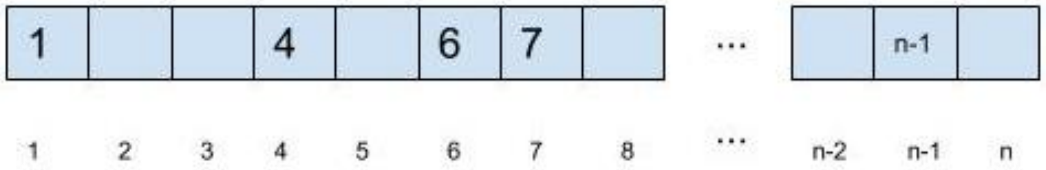


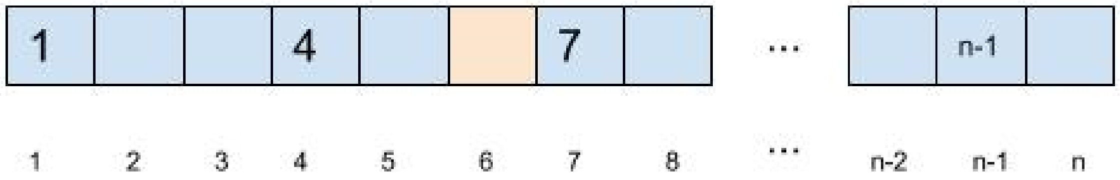
- Time Complexity
  - **<u>Direct indexing</u>**
  - O(1)

GEORGETOWN UNIVERSITY

Motivation: Simple Example

- Removal
- Remove 6
- Time Complexity
  - Direct Indexing
  - O(1)



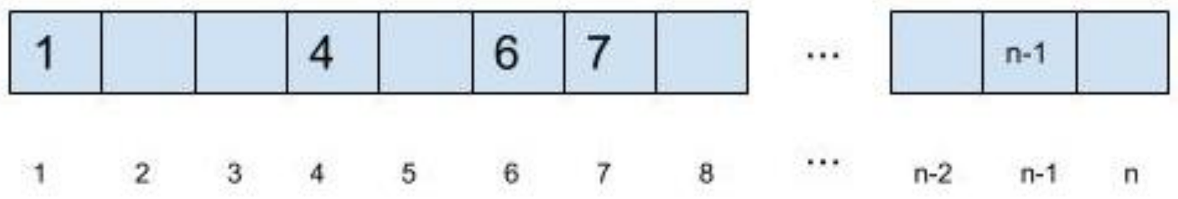


GEORGETOWN UNIVERSITY

Motivation: Simple Example Analysis

- How are we able to attain such a time complexity?
  1. It is known, *apriori*, where each item is (to be) stored
  2. Direct indexing: indexing is accomplished in constant time

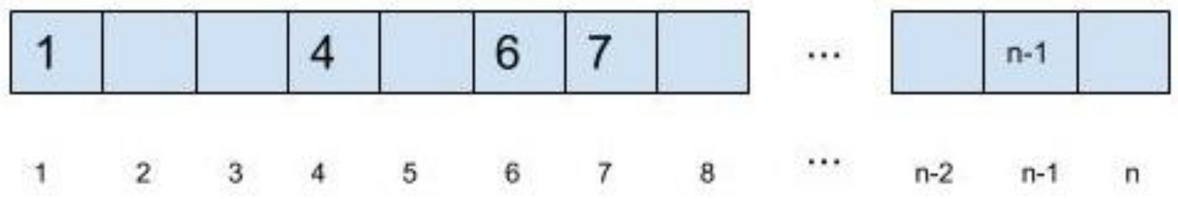
- We know where to go, and we can get there fast!



GEORGETOWN UNIVERSITY

Simple Example: How?

- Direct indexing is no mystery. But how did we know, *apriori*, where each item is (to be) stored
  - The value stored was simply the index!



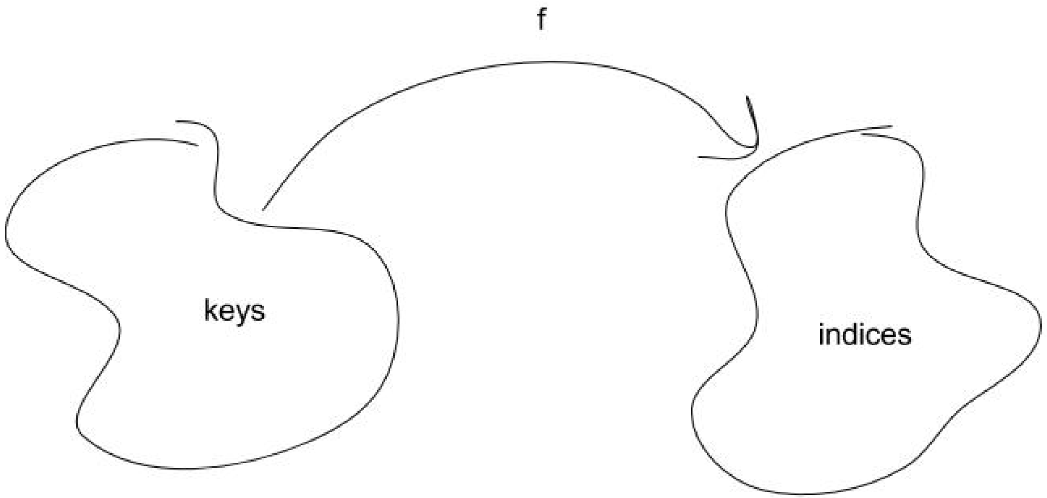
- This works well if we are storing integers, but what about non-integer data types or values that are not within a good indexing range…?

GEORGETOWN UNIVERSITY

Direct Indexing with non-integer types

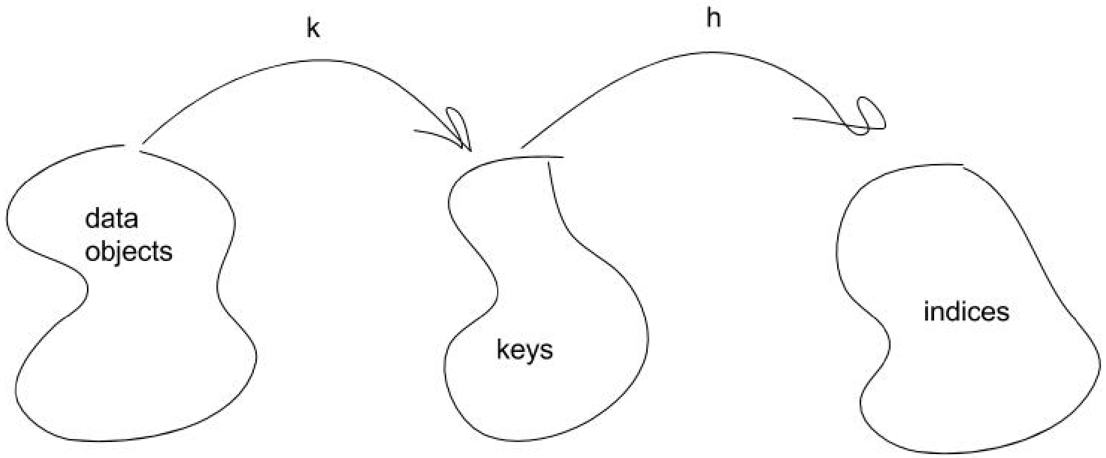
- In our simple example, the key (which is the data itself) is also the index.
  - That is, each data value, directly maps to an appropriate index.

- Solution general case: find a function f that maps from the set of keys to a set of indices.



The hash function

- Remember: a key uniquely identifies a data object
  - Let k be a one-to-one correspondence that maps from data objects to keys
  - $k: dataObjects \rightarrow keys$
- A **hash function** is a function that maps from a set of keys to a set of indices
  - $\mathrm{h}: keys \rightarrow indices$

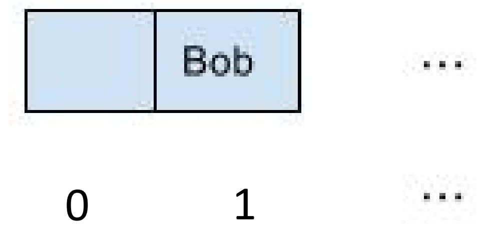


Hash Example

- Assume we wish to store student information in an array.

- Example data object
  - (fname, lname, studid, age, <img>, …)
  - (Bob, Barker, 1111111, 18, <img>, …)

- Example key: studid
  - $k\colon \text{studentObjects} \to \text{studentIDs}$
  - $k(Bob) = 1111111$



- Example hash
  - $\text{h}\colon \text{studentIDs} \to \text{indices}$
  - $h(x) = x \bmod 1000$

$$h\big(key(Bob)\big) = 1$$

Domain of Keys

- It is important to identify and define the <u>domain</u> of keys, K, of a hash.

- Characteristics of the keys will largely determine the type of hash to be used.

- In many instances the total possible keys $|K|$ is larger than the number of actual keys to be stored $\mathrm{N} \subset K$, $|\mathrm{N}| < |K|$
  - For example, the total number of students to be stored may be 10,000, but the total number of possible student id values may be 0 – 999999, numbering 10,000,000
  - In our discussion we *may* assume these values are the same, ***<u>which is often not the case.</u>***

Hashing: Design Concerns

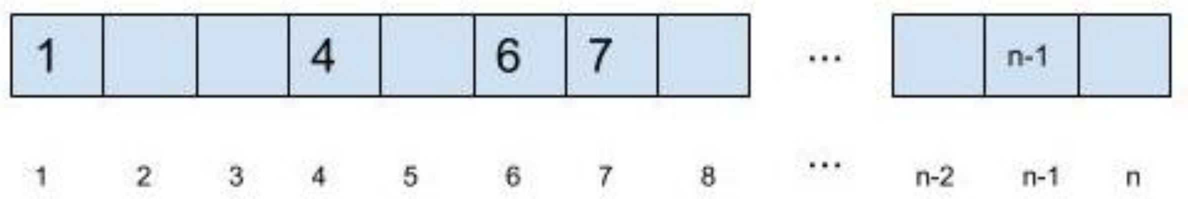
- Time complexity for search, insert and removal is constant!
  - Why not always use a hash?!

- Design Concerns
  1. Space:
     1. Cardinality of keys, $|K|$, is likely large. The space requirements for a hash is not necessarily bound by the size of the input.
     2. |N| may or may not be known. Is size static or dynamic?
  2. What happens if two different values get mapped to the same index?
     1. Collision
  3. Finding a hash function that mitigates these concerns is hard
     1. Searching over a family of hash functions may not be tractable
     2. Time complexity of evaluating the hash, eg computing h(k), may be a concern

GEORGETOWN UNIVERSITY

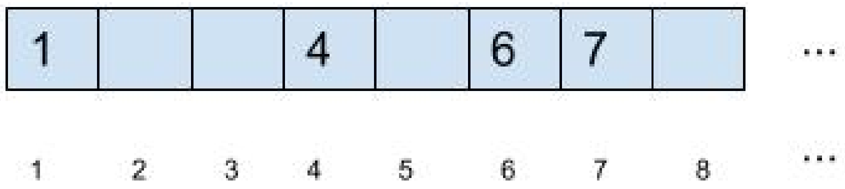
Hashing: Space concerns

- Example 1. Reasonable Index Range.
  - Store up to 1000 values *within range 1 – 1000.*
  - More generally: store up to n values within range 1 – n.
    - Note here the size of the input may be up to n numbers, thus we can bound the memory constraints in terms of n, the size of the input.
  - Use simple hash: h(i) = i
    - Size of input (potentially n) , size of space requirements O(n)



Hashing: Space concerns

- Example 2. Unreasonable Index Range.
  - Store up to n values within *unknown* range, eg, |K| is large.
    - Size of input is n, how big of an array is needed
  - Using simple hash from Ex. 1, is not efficient
    - Use simple hash: h(i) = i
      - Size of input (potentially n) , size of space requirements …?
    - In this simple case, space requirements depend on the value of the input, and not the size of the input.

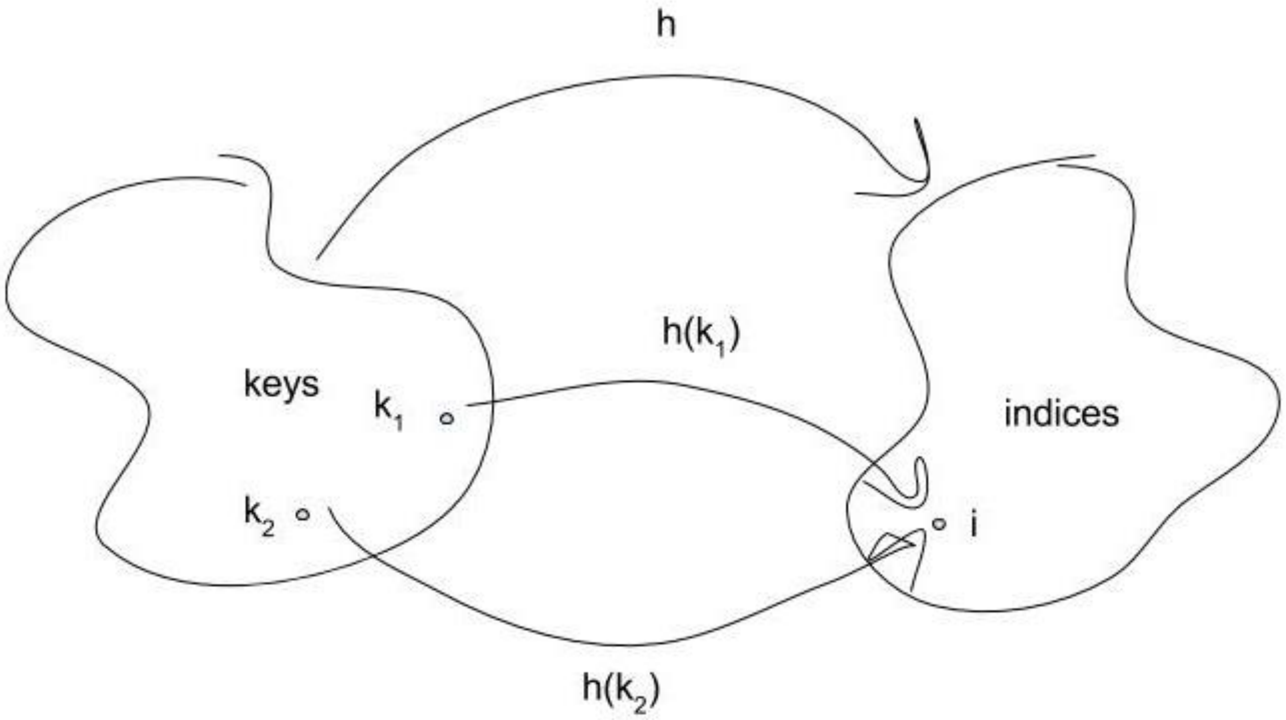


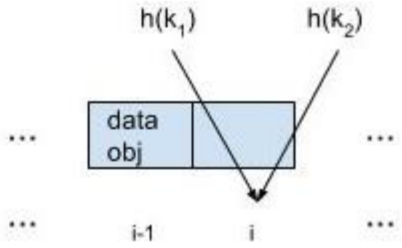
GEORGETOWN UNIVERSITY

Hash: Collision Concerns

- If a hash function is not a one-to-one correspondence, then a collision is possible

- A collision occurs when a hash function maps two different key values to the same index.





Finding a Desirable Hash Functions

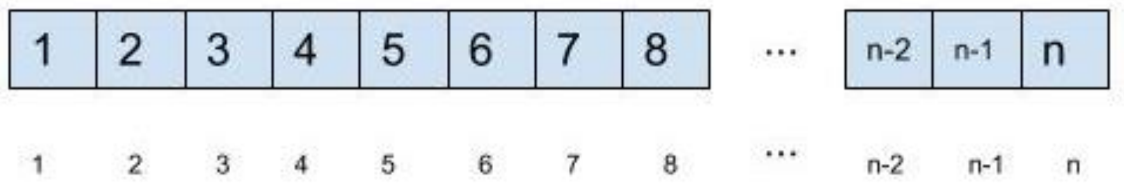
- Desirable characteristics of a hash function
  - Space efficient. If n items to store, O(n) space.
  - Minimize collisions
  - Hash computation is fast

- A **hash table of size m** is an array of size m that uses a hash function for indexing (for searches, inserts and removals)

Addressing space efficiency

- Our simple hash example is a poor choice
  - h(i) = i
  - This depends on the value of keys and not the number of keys.

- Optimally. If there are n items to store, use an array of size m=n.
  - Finding a hash function that maps each item perfectly (without collision) and has no wasted space is ***very difficult*** in the general case.

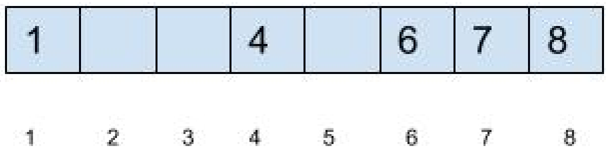


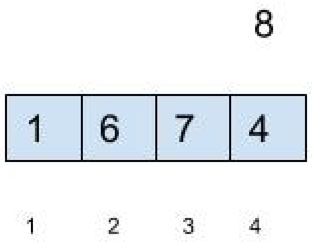
GEORGETOWN UNIVERSITY

Addressing space efficiency

- Intuitively, *there is a tradeoff between the size of the hash table and frequency of collisions*.

- Intuitively, assume indices were randomly assigned to n different keys, the probability of collision would increase if the range of indices was reduced (if the array was smaller)





GEORGETOWN UNIVERSITY

Addressing Collisions

- Collisions are often inevitable.

- Assume $(n-1)m+1$ keys. $|\mathrm{K}| = (n-1)m+1$.
  - Table size m

- Notes:
  - Since there are m locations for $(n-1)m+1$ items, there will be a set of n elements that hash to the same location, (the pigeon-hole principle).
    - If m is less than the number of items to store, there will be collisions!

Some Hashing Schemes

- Division Method
- Folding Method
- Mid-Square Method
- Radix Method
- Universal Hashing
- Perfect Hashing
- Double Hashing

Hash: Division Method

- If nothing is known about the keys prior, the division method is a commonly used solution.
- Use simple int interpretation of full binary representation of k, Hashtable of size m

$$h(k) = k \bmod m$$

Variant: keys of vector of ints

Pros: fast, maps to valid indices

Cons: unknown (without more information about distribution of keys). No assurances about collisions.

Folding Method

- Method.
  1. key is partitioned,
  2. each partition is manipulated,
  3. then the results are aggregated together (folded together) to produce a final index

- Example.
  – SSN key.
  – Partition into 3 sections.
  – Add three parts
  – Perform division on resulting scheme (table size m = 1000)

Key k: 123- 45-6789

Partition and Add: 123 + 45 + 6789

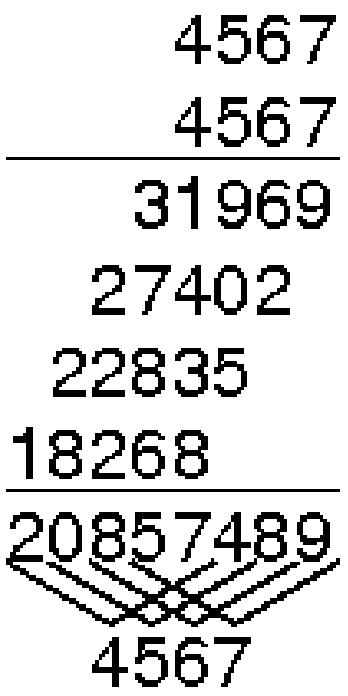
Sum: 6,957

index =
6,957 mod 1000 =
957

- Pros.
  – Provides means to incorporate more digits into index computation.

Hash: Mid square approach

- Mid square approach
  - Interpret binary sequence of key as an int
  - Hash function
    - Square the key value
    - Use the r-inner bits as the index
  - Intuition
    - Pro: All digits will affect the innermost bits of the squared value.
    - Generally good uniform distribution
    - Con: must compute square



- Example:
  - Key 4567
  - R= 2
  - Use 2-inner bits of square. 57 is index

GEORGETOWN UNIVERSITY

Radix Method

- Radix transform.
  - Rescale range of values by changing the number system.
  - Assume key k is numeric.
  - Map number to different radix (base)

- Example.
  - Assume k = 345, and table size m = 100
  - Change radix to 9
  - $k = 345_{10} = 423_9$
  - h(423) = 423 mod 100 = 23.

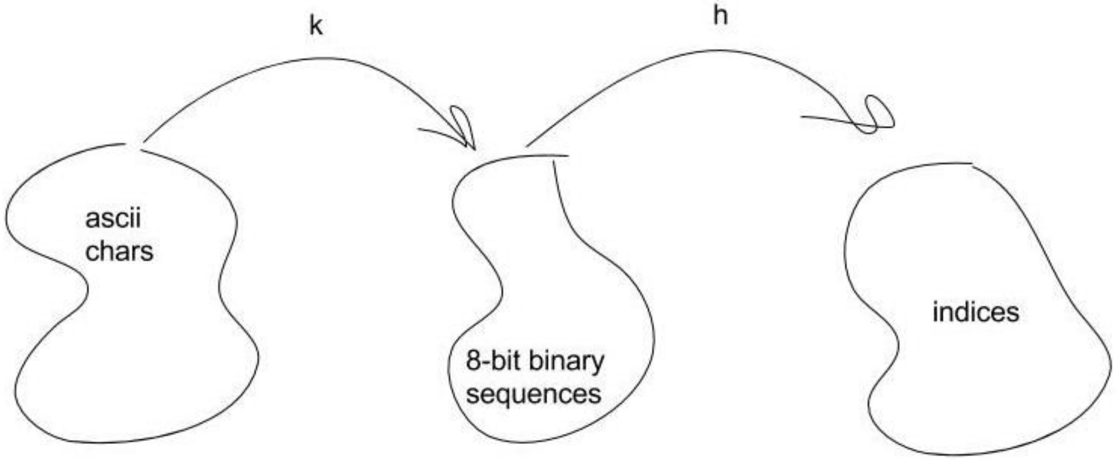
Simple Hashing Schemes

- Simple Schemes.
  - Division Method
  - Folding Method
  - Mid-Square Method
  - Radix Method

- Observations.
  - Provide an means to mathematically map keys to index range.
  - These mapping schemes are
    - Easy to implement
    - Hash is fast to compute
    - But make no assurances about collisions (unless keys are known *apriori*)

Designing a Hash. Case Study Chars

- Storing ASCII characters
  - chars are stored in 8 bits, thus there are 256 unique chars to store
  - 256 unique data objects
    - Not many, lets simply create a hash table of size 256
  - What is a good key?
    - **Trivial key mapping**. Each char has a unique binary encoding … which easily can be interpreted as a non-negative integer using polynomial expansion … lets use that!
  - Hash is simply the int interpretation of the keys binary value
    - **Trivial Hash.** one-to-one correspondence and space efficient!



Designing a Hash. Case Study Strings

- Store a set of n strings
  - String: Simply a sequence of chars
  - Key ideas
    - Strings are unique based on uniqueness of each char at each location
    - Simple concatenation of binary sequence of chars
  - Hash ideas
    - (Bad) Idea 1: Using int interpretation of FULL binary sequence
      - Would provide for no collisions, but at what cost!
        - » Assume: Each char may use up to 8-bits. Longest string will be 25 chars long
        - » Possible indices needed (for no collisions): 2^{200}

Designing a Hash: Case Study String (cont)

- Idea 2: Simple design scheme.
  - *Fix the table size* to something reasonable, m = 1000.
  - Use **Folding Scheme**. Sum numeric interpretation of each characters
    - hash function: h(string) = sum mod 1000
  - Alleviates space concerns, but may result in collisions.
  - Note here N < |K| is likely and so allocating |K| spaces may be unnecessary and impractical

  - <u>Observation</u>: may not map to range 0 – 999 very uniformly. May result in more collisions that desired.
    - EG: all of the following strings would map to the same key
      - » az, za, by, yb, cx, xc, …

Designing a Hash: Case Study Strings

- Universal
  - Randomly construct b-u random matrix
    - $\mathrm{b} = \log_2 m$
    - u is number of bits for keys (200 bits)
  - Randomly select k numbers and use variant.

- Perfect (using O($|K|^2$) space approach)
  - Choose $\mathrm{m} = |K|^2 = 2^{200^2}$ (not practical!)
  - $\mathrm{b} = \log_2 2^{200^2} = 40000$

- Double Hash
  - (often) an efficient solution

- Another Method. **Cichelli's Method.**
  - Searches for a Hash Map that works well.
  - Search time may be expensive.
  - See Readings and HW questions.

GEORGETOWN UNIVERSITY

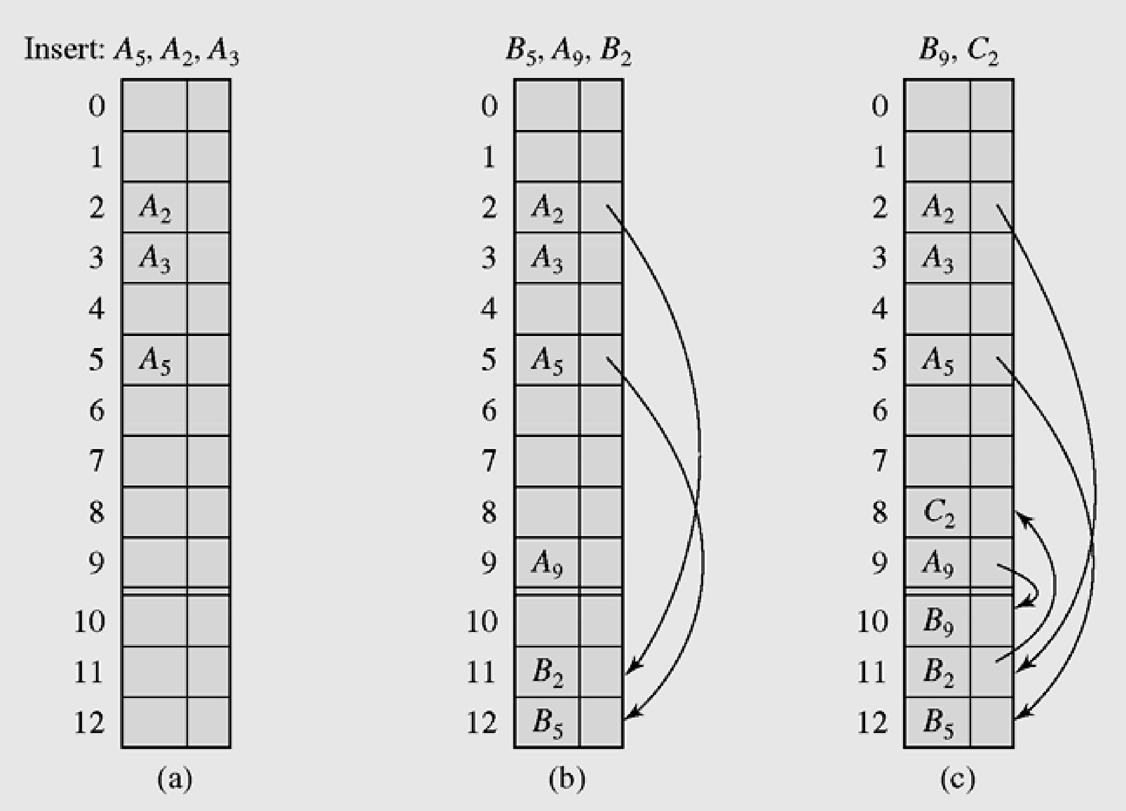
Collision Resolution

- In some instances collisions may be hard to avoid.

- Having an efficient resolution scheme is important.
  - Annex / Cellars
  - Probing
  - Chaining

Collision Resolution: Annex or Cellar

- Example table of size 10
- Scheme: reserve c spots to end of array; designate as cellar. Store collisions there sequentially.
- Worst Case Complexity: O(c)
- Cons:
  - Cellar size = c is fixed
    - may fill up
    - Is unordered and generally large



GEORGETOWN UNIVERSITY

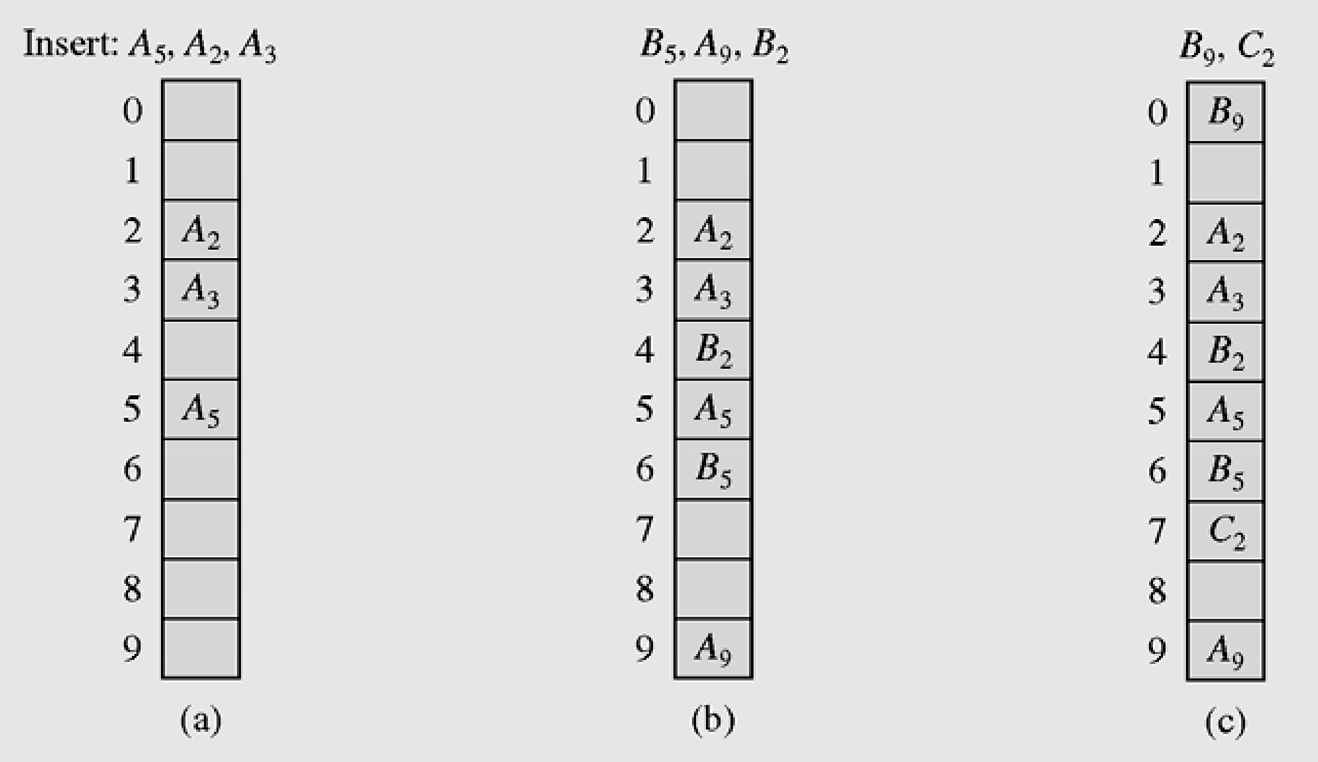
Collision Resolution Scheme: Probing

- **Linear Probing**

- Scheme: rather than store collisions in cellar, store them in an empty location near correct hash index. Use linear probe to find nearby empty locations.

- Linear probe is generally a simple sequential scan (with mod wraparound)
  - Ex: h(k) , h(k) + 1, h(k) + 2, …

- Complexity analysis:
  - Dependent on load of table
  - **<u>Load</u>**, λ, is percentage of occupied locations
  - Average Case: $O\left(\frac{1}{1-\lambda}\right)$



GEORGETOWN UNIVERSITY

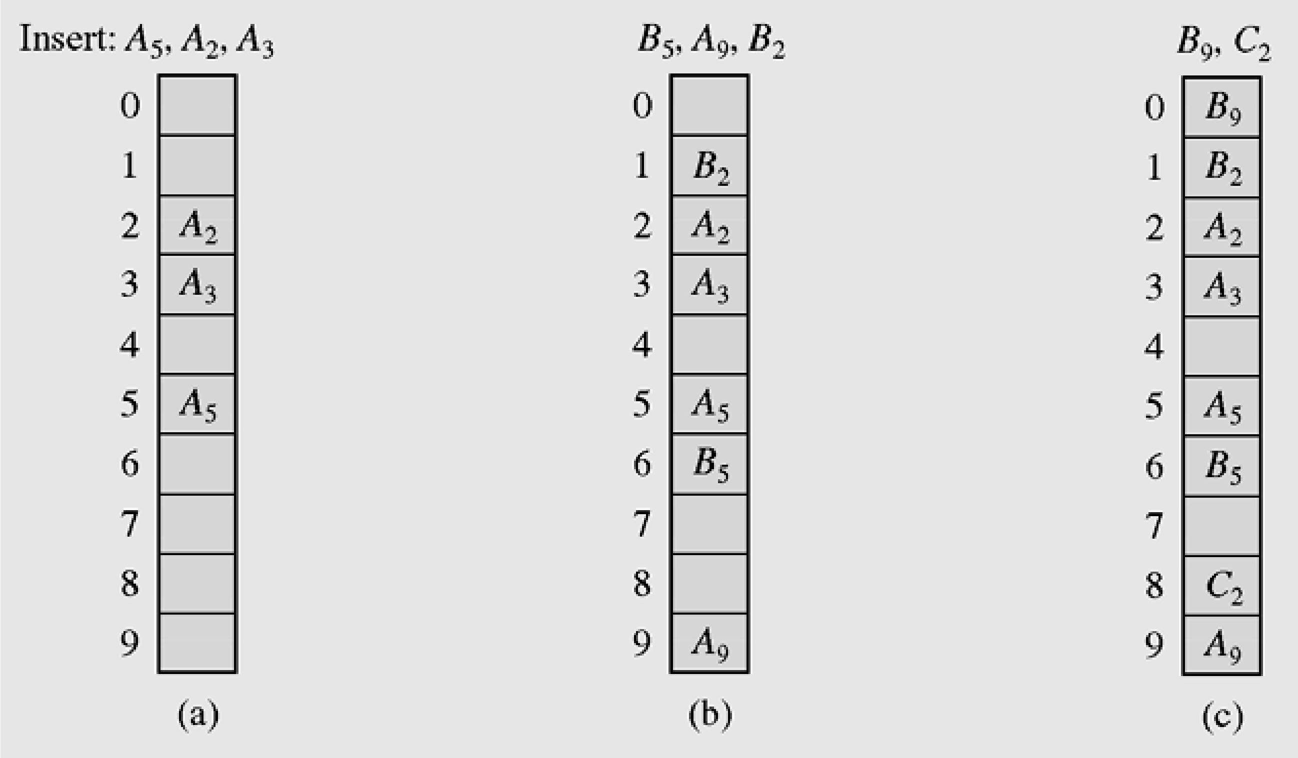
Collision Resolution Scheme: Probing

- **Quadratic Probing**

- Linear probing may suffer if keys are not uniformly distributed. “Clustering” in some regions of the table will occur which will increase the number of overall collisions.

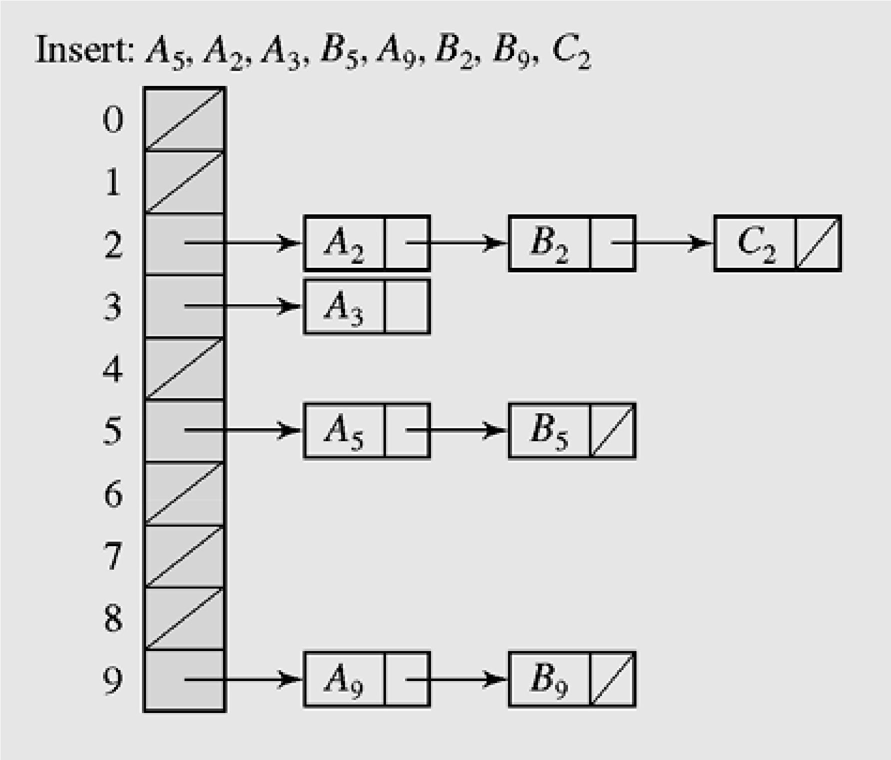
- Scheme: Search for empty spaces, further away.
  - h(k), h(k)+1, h(k)+4, h(k) + 9, …

- Complexity:
  - Must be sure to traverse indices without repetition.
  - This is assured if m is prime.



Collision Resolution: Chaining

- Each Array entry is a linked list
- Collision is implicitly handled by adding to top of list.
- Pros:
  - Dynamic size,
  - Static size issues resolved: cellar overflow, or table overflow (probing)
- Complexity:
  - Conceptually better than cellar as the collision space is organized by original hash entry
  - Assume c is number of total collided items over m buckets. Average case: O(c/m)
  - Practical Concerns: memory not contiguous, may have disk access delays
- Alternative (to linked list) approaches
  - Implement Hash table, where each bucket is a B-Tree
  - Implement Hash table, where each bucket is a hash table



GEORGETOWN UNIVERSITY

Collision Resolution Schemes

- Cellar
  - Static Size
  - Sequential search in cellar if collision

- Probing
  - Static size
  - Searching is done locally if collision
  - Efficiency highly dependent on average load of table

- Chaining
  - Dynamic size
  - Possible delays related to non-contiguous allocation
  - Organized search if collision

Avoiding Collisions: Statistical Perspective

- If all the keys are known *apriori*, then we can construct a simple hash that avoids all collisions. However, this is not often the case.

- Some CS problems are hard (eg collision-free hashing when little is known about the keys *apriori*), and finding an *optimal solution* is impractical.
  - Sometimes its more appropriate to find a *good* solution (with high probability) fast.
  - Statistical approaches: Quantify probability of poor result.
  - Rather than trying to avoid all collisions, quantify (*minimize*) how often they might occur.

- Universal Hash Idea. (Monte Carlo Scheme)
  - Assume n items are assigned indices randomly by h.
  - We can statistically bound the number of collisions, if we construct the hash in a "random" sense.
    - We can choose the size of m to bound the number of expected collisions.

GEORGETOWN UNIVERSITY

Uniform Hash

- Having a hash function that uniformly assigns keys to buckets is desirable
  - Reduces “clustering” and collisions

- A uniform hash has a uniform probability of collision

$$P(h(k_i) == h(k_j)) = P(C_{ij}) \leq \frac{1}{m}$$

Universal Hashing

- A **Universal Class of Hash Functions H** is a set of hash functions with the following bound on collisions.

- When any hash function $h \in H$ is chosen randomly from H, the probability of collision of any two keys (key i and key j) is bounded as follows:

$$P(h(k_i) == h(k_j)) = P(C_{ij}) \leq \frac{1}{m}$$

Observe: The number of collisions of key i with ***any other*** key can be bounded (proof upcoming)

- Constructing a hash function with this property is not as difficult as it might seem (given some assumptions about the data).
  – The crux is having the means to randomize the selection.

GEORGETOWN UNIVERSITY

Universal Hashing: Expected Number of Collisions

- Similar to previous slide (here we take expectation of Boolean variable collision.)
- The expected number of collisions between x and other elements in S is at most N/M
- Proof …
- Since $P(C_{ij}) \leq \frac{1}{m}$
- Let C_{ij} = 1 , when i and j collide.
- Let C_i be the total number of collisions for i. $\mathrm{C_i} = \sum_{j=1}^{n} C_{ij}$
- We assume $E[C_{ij}] \leq \frac{1}{m}$.
- Thus $E[C_i] = \sum_{j}^{n} E[C_{ij}] \quad \leq \frac{n}{m}$. , by linearity of expectations

GEORGETOWN UNIVERSITY

What is a class or family of functions?

- Example: Consider a set of functions F
  - $F = \{f_1, f_2, \ldots, f_{100}\}$, where $f_i(x) = \frac{2x}{i}$
- Observe F is a class of 100 distinct functions, which vary based on some parameter $i \in \{1,2,\ldots,100\}$.
- If we can randomly select i, then we can randomly select a function in F.
  - Crux: find a family that meets the universal property!

Designing a Universal Hash Family (Matrix Method)

- Assume
  - keys are u-bits long
  - Table size m is a power of 2, m = 2^b

- Define hash function h in terms of random 0-1 matrix H, that is b x u.

$$h_H(k) = Hk$$

- Example

M = 4
Keys: 3-bits

$$\begin{array}{ccc} H & k & Hk \\ \begin{bmatrix} 1 & 0 & 1 \\ 0 & 0 & 1 \end{bmatrix} & \begin{bmatrix} 1 \\ 1 \\ 0 \end{bmatrix} = & \begin{bmatrix} 1 \\ 0 \end{bmatrix} \end{array}$$

Pro. This is a universal hash, *ie*, $P(h(k_i) == h(k_j)) = \frac{1}{2^b} = \frac{1}{m}$
Con. Hash computation cost: $\mathrm{O}(\log_2 \mathrm{m} \times \log_2 |\mathrm{k}|)$

Universal Hash Familty: Division Method

- Represent keys as d-dimensional vectors of integers
  - Rather than vector of binary values,

- Key example. x = $[x_1, x_2, \dots, x_d]^T$

- Hash Table
  - Size: m
  - Choose d random non-negative integers, $\mathrm{r}_{\mathrm{i}\in\{1,\dots,d\}} < m$
  - $h(x) = [r_1, r_2, \dots, r_d\,][x_1, x_2, \dots, x_d]^T = \left[\sum_{i=1}^{d} r_i x_i\right] mod\ m$

- Notes:
  - Computation of key (and storage of hash parameters) is seemingly less than matrix method
  - If m is prime, then a universal hash is guaranteed.

Perfect Hashing

- A perfect hash is a hash function where search, insert and removal are all O(1).
  - IE, no collisions (or constant bound on collisions).

- If the set of keys (to be inserted) is known apriori, finding a perfect hash may be practical.

- What if we do not know all keys apriori?
  - One solution: repeatedly pick a (random) universal hash until it is perfect.
    - But how long will this take?
    - If we select $\mathrm{m} = |K|^2 = n^2$, the probability of no collisions is greater than $\frac{1}{2}$
    - Cost: O($|K|^2$) space

GEORGETOWN UNIVERSITY

Perfect Universal Hash: Quadratic Space Method

- Conjecture: Let h be a draw from a Universal Hash family. If $\mathrm{m} = |K|^2 = n^2$, then the probability of no collisions is greater than ½ .

- Proof
  - Let C_{ij} denote a collision between key i and j
  - The number of pairwise collisions possible in K, where |K| = N is n choose 2, $\binom{n}{2}$
  - Recall $P(C_{ij}) \leq \frac{1}{m}$
  - Thus, the $P(\exists_{ij}\, C_{ij} == 1) \leq \frac{\binom{n}{2}}{m} = \frac{\binom{n}{2}}{n^2} = \frac{\frac{n!}{2!(n-2)!}}{n^2} = \frac{n \quad (n-1)}{2n^2} = \frac{n \quad (n-1)}{2n^2} = \frac{(n-1)}{2n} \leq \frac{1}{2}$

GEORGETOWN UNIVERSITY

Hash of Hashes Scheme: Universal and Perfect!

- Previously we discussed using a O($|K|^2$) space approach to attain a perfect hash (by repeatedly, selecting a random universal hash)
  - Good, but we can do better than O($|K|^2$) space!

- Scheme: create a hash table of hash tables!
  1. Create a universal hash of size $|K|$ (or N , the number of items to insert if we know N *apriori*)
     - This may result in some collisions, c_i, for each bin i in the table, which is OK.
  2. For each bin i, create a hash function using the O($|K|^2$) space approach , where $|K|^2 = |c_i|^2$
     - Intuition: the number of collisions in each bin should not be very much
     - Note: the total $\sum_i |c_i|^2$ can be bounded linearly by N with high probability.
     - Result: O($|K|$) space

GEORGETOWN UNIVERSITY

Perfect Universal Hash: Linear Space

- Conjecture: Assume a universal hash h is chosen where m = n. Let c_i be the number of collisions in bucket i , then $\mathrm{E}\left[\sum_i c_i^2\right] < 2n$, thus the total space is linearly bound.

- Proof
  - $\mathrm{E}\left[\sum_i c_i^2\right] = E\left[\sum_i \sum_j C_{ij}\right] = N + \sum_i \sum_{\substack{j \\ j \neq i}} E[C_{ij}] \ = \ \ldots$

  The rest of this proof is left as an exercise.

RE-Hashing

- Sometimes the choice of hash or size of hashtable is poor, and should be changed.
  - In general this results in a complete reconstruction of the hash table, which is slow: O($|K|$)

Hash Summary and Time Complexity

- The crux – finding a hash with a good balance of space requirements and minimal collisions.

- Hash table of size m allocation: O(m)

- Search, Insert, Remove (assuming constant hash computation)
  - Without collision: O(1)
  - With collision (cases may be): O(c) or $O\left(\frac{c}{m}\right)$ or $O\left(log\left(\frac{c}{m}\right)\right)$ or …
    - If you expect many collisions, employ an efficient collision resolution scheme (and/or consider increasing the size of your table)

- Using universal hashing we can statistically bound the number of collisions and space requirements resulting in an Perfect, Linear Space, Hash!

GEORGETOWN UNIVERSITY

Bonus: Radix Sort

- Sort items in list, one digit at a time using a (simple) hash with chaining

- See supplemental PPT for animated example.

Algorithm 1

Require: Array is an array of ints of length n.
  function RADIXSORT(int* array, int n)
    $radix \leftarrow= 10$
    $digits \leftarrow= 10$ // max num digits in an int
    $q \leftarrow$ intialize an array of FIFOqueues of length digits
    for d from 0 to $digits$ **do**
      for i from 0 to $n-1$ **do**
        q[dth digit of array[i]].enqueue(data[i])
      $k \leftarrow 0$
      for j from 0 to $radix - 1$ **do**
        while $\neg$ q[j].isEmpty() **do**
          array[k++] $\leftarrow$ q[j].dequeue()