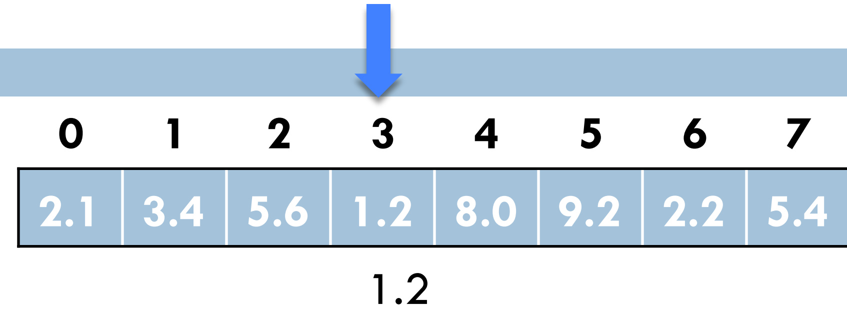
CSC230

Intro to C++ Lecture 19

How to find stuff?

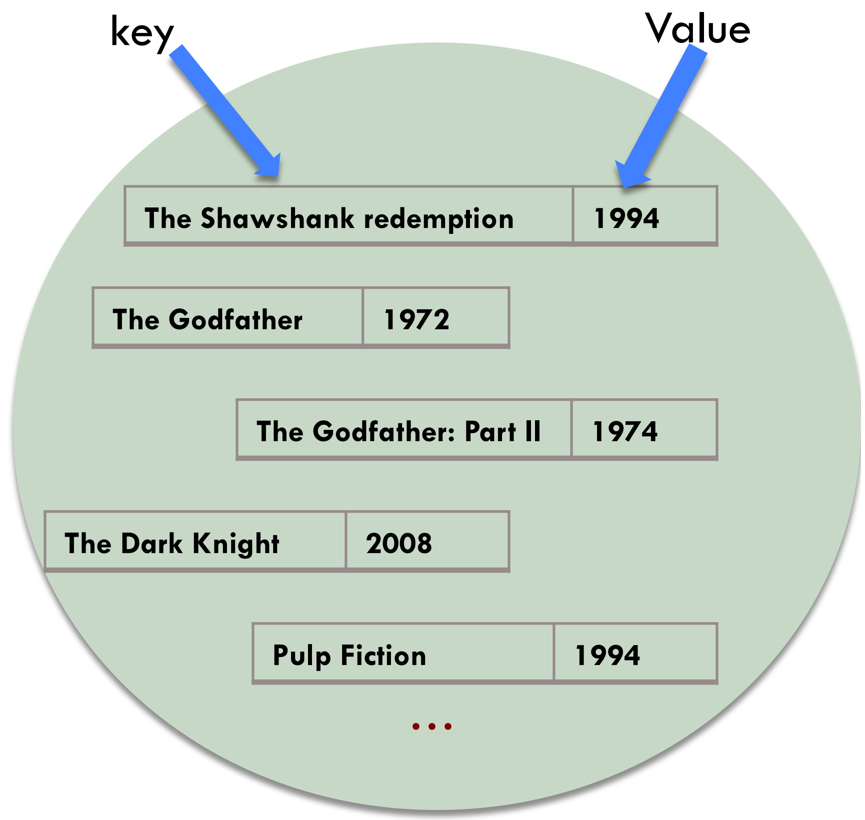


Array:

- Given an index value
- Retrieve the corresponding item

Map/Dictionary:

- Given an key value k
- Return associated map[k] value



How to organize the data of Map/dictionary ?

Array?

- Entries are ordered by Key values
  - In **worst** case, after (log n) comparisons, we will find the entries. More details in searching lectures
- Entries are not ordered
  - On **average**, need to compare n/2 entries
  - In worst case, need to compare n entries

Linked list?

- Entries are ordered by key values
  - In worst case, needs n comparisons
  - On average, needs n/2 comparisons
- Entries are not ordered
  - In worst case, needs n comparisons
  - On average, needs n/2 comparisons

Slow !!

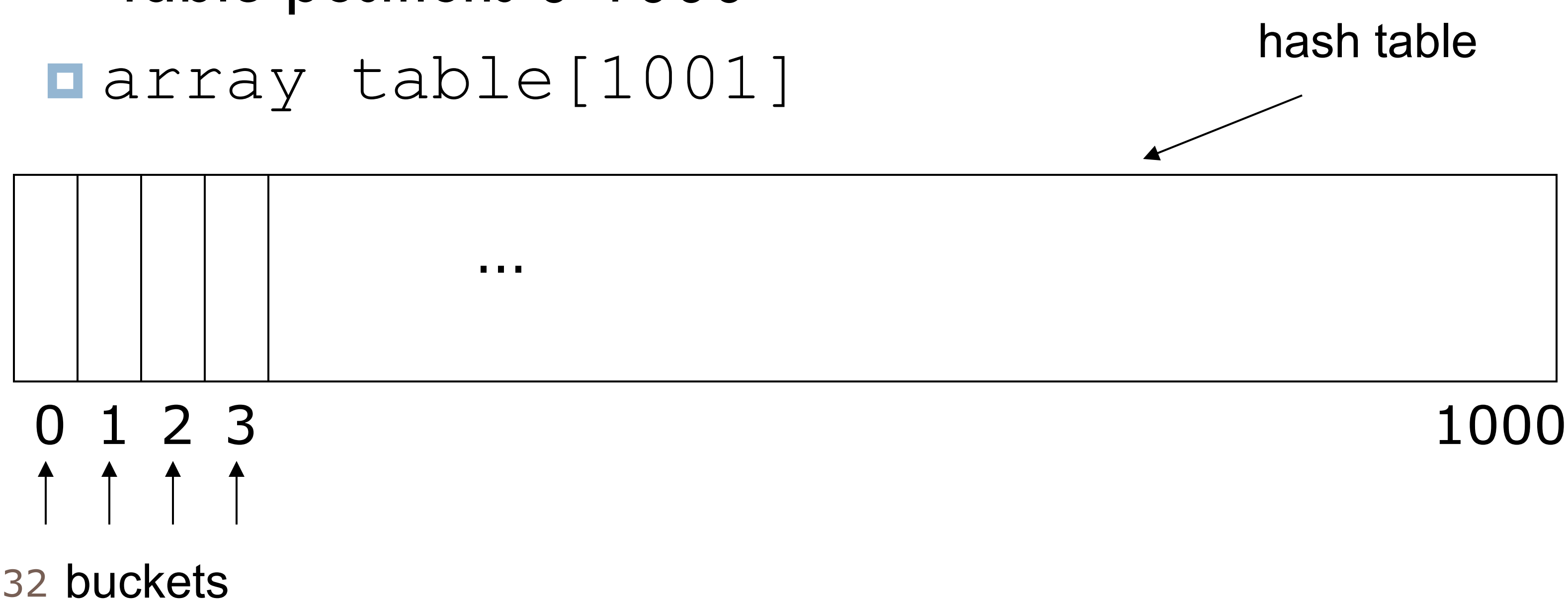
Hash Tables: Basic Idea

- Use a key (arbitrary string or number) to index directly into an array – O(1) time to access records
  - A["kiwi"] = "Australian fruit"
  - Need a *hash function* to convert the key to an integer

	Key	Data
0	kim chi	spicy cabbage
1	kreplach	tasty stuffed dough
2	kiwi	Australian fruit

Example

- Dictionary Student Records
  - Keys are ID numbers (951000 - 952000), no more than 100 students
  - Hash function: h(k) = k-951000 maps ID into distinct table positions 0-1000
  - `array table[1001]`



Hashing

Idea:
- Save items in a key-indexed table (index is a function of the key)

Hash Function:
- Computing array index from the key

hash("is") = 1

hash("but") = 5

hash("this") = 5

0	life
1	is
2	good
3	
4	!
5	but
6	isn’t
7	
8	cafe

Issues with Hash Function:
- Time complexity: trivial
- Space complexity: trivial
- **Collision**: two different keys mapped to the same index
  - Need Algorithms and data structure to handle it

Hash Function

Goals: scramble the keys uniformly
- Each index is equally likely for each key
- Time complexity is low

Ex. Phone number:
- First three digits. ✗
- Last three digits. ✔
- How about the middle three digits?

609 = trenton area, 732 = central NJ

Ex. Social Security Number:

SSN Area Number	Location
001-003	New Hampshire
004-007	Maine
008-009	Vermont
010-034	Massachusetts
035-039	Rhode Island
040-049	Connecticut
050-134	New York
135-158	New Jersey
159-211	Pennsylvania

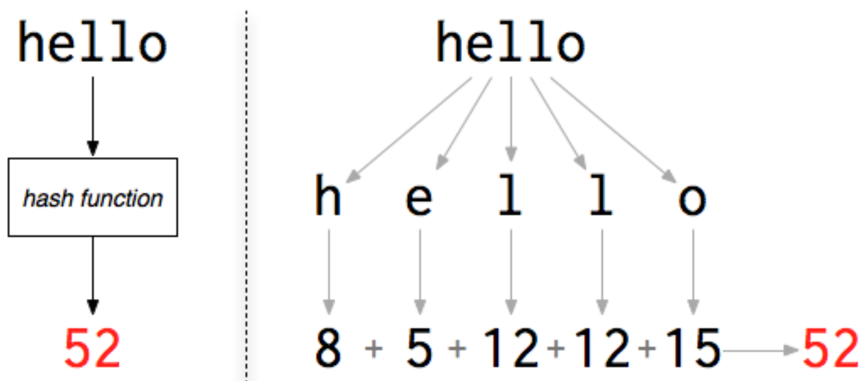
Hash Function

A Hash Function scrambles the keys uniformly

- Map the key value to an integer
- The integer is in the range of [0, tableSize - 1]

Usually, hash function is provided by user

Ex. Strings



Keeping hash(k) within the range of table size

- hash(k) % tableSize

Table Size ?

Ideal Situation:

- Each pair of (key, value) has its own entry
- Example: New Jersey Vehicle Plates: AZY 55M, CCA T23 would require how many table entries?
  - At least 36^6
  - Not all the plate numbers used
  - We often need a (small) subset of keys

Practical Situation:

- Table size is larger than the expected entries
- If it is smaller than the number of key values, **COLLISION**

Make the table size be **Prime number**

- More discussion later

How to resolve collision ?

Table Size < the number of Key Values:

- According to the principle, collision is unavoidable

Table Size >= the number of Key Values:

- If hash function is well designed, no collision
- otherwise, **COLLISION**
  - Phone number example: first three digits, last three digits

Make the table size be **Prime number**

- It may **reduce** collision chance
- More discussion later

Closed Hashing

- Associated with closed hashing is a *rehash strategy*:

  "If we try to place x in bucket $h(x)$ and find it occupied, find alternative location $h_1(x)$, $h_2(x)$, etc. Try each in order, if none empty table is full,"
- $h(x)$ is called *home bucket*
- Simplest rehash strategy is called *linear hashing*

$$h_i(x) = (h(x) + i) \% D$$

- In general, our collision resolution strategy is to generate a sequence of hash table slots (probe sequence) that can hold the record; test each slot until find empty one (probing)

Collision resolution: open addressing/closed hashing

- Open addressing. [Amdahl-Boehme-Rocherster-Samuel, IBM 1953]
  - When a new key collides, find alternate location in the array, and put it there

  - Linear probing
    - The Interval between probes is fixed – often at 1
  - Quadratic probing
    - The interval between probes increases linearly
  - Double hashing
    - The interval between probes is fixed, calculated by another hash function

Linear probing

- Primary clustering

Certain data patterns lead to many collisions, linear probing leads to clusters of occupied areas in the array

Key	value
0	occupied
1	
2	occupied
3	occupied
4	occupied
5	
6	
7	occupied
8	...

Example Linear Hashing

- D=8, keys *a,b,c,d* have hash values h(a)=3, h(b)=0, h(c)=4, h(d)=3

- Where do we insert *d*? 3 already filled
- Probe sequence using linear hashing:

  $h_1(d) = (h(d)+1)\%8 = 4\%8 = 4$

  $h_2(d) = (h(d)+2)\%8 = 5\%8 = \mathbf{5}^*$

  $h_3(d) = (h(d)+3)\%8 = 6\%8 = 6$

  etc.

  7, 0, 1, 2
- Wraps around the beginning of the table!

0	b
1	
2	
3	a
4	c
5	**d**
6	
7	

Operations Using Linear Hashing

- Test for membership: *findItem*
- Examine $h(k)$, $h_1(k)$, $h_2(k)$, …, until we find k or an empty bucket or home bucket
- If no deletions possible, strategy works!
- What if no empty bucket?
  - **Chaining**

Quadratic probing

- Primary clustering

Quadratic probing takes larger and larger steps, which tends to spread out the data across the array

Key	value
0	occupied
1	
2	occupied
3	occupied
4	occupied
5	
6	
7	occupied
8	…

Improved Collision Resolution

- Linear probing: $h_i(x) = (h(x) + i) \% D$
  - all buckets in table will be candidates for inserting a new record before the probe sequence returns to home position
  - clustering of records, leads to long probing sequences
- Linear probing with skipping: $h_i(x) = (h(x) + ic) \% D$
  - c constant other than 1
  - records with adjacent home buckets will not follow same probe sequence
- (Pseudo)Random probing: $h_i(x) = (h(x) + r_i) \% D$
  - r_i is the i^{th} value in a random permutation of numbers from 1 to D-1
  - insertions and searches use the *same* sequence of "random" numbers

How to find/remove?

- Given linear or quadratic clustering, how to find a (key, value) pair?
  - Hash the key
  - If the (key, value) pair is not at hash(key), search linearly or quadratically
    - Find it or
    - An empty location or
    - Search the whole array

- How to remove the pair ?
  - First find the pair in the array
  - Mark the location as “removed”. You must use a special thing to label it; otherwise, the insertion will be messed up, why?

Double hashing

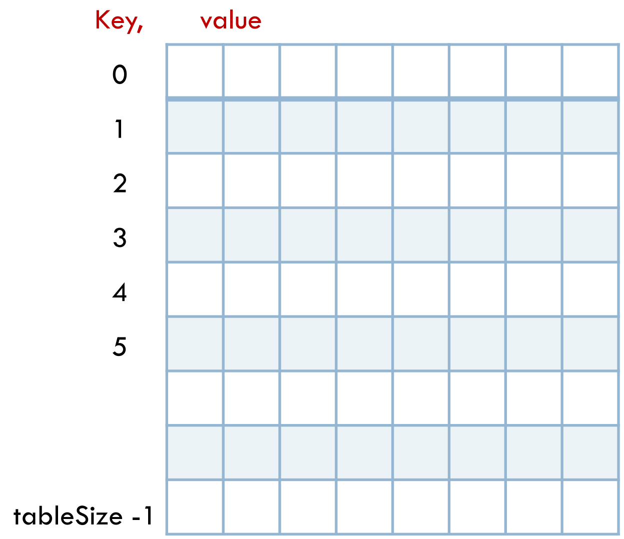
- Two hashing functions, $h_1(k)$ maps key k to an array location, $h_2(k)$ is the interval between probes
    - First check $h_1(k)$
    - If occupied, then check $h_1(k) + h_2(k)$
    - If occupied, then check $h_1(k) + 2*h_2(k)$
    - If occupied, then check $h_1(k) + 3*h_2(k)$

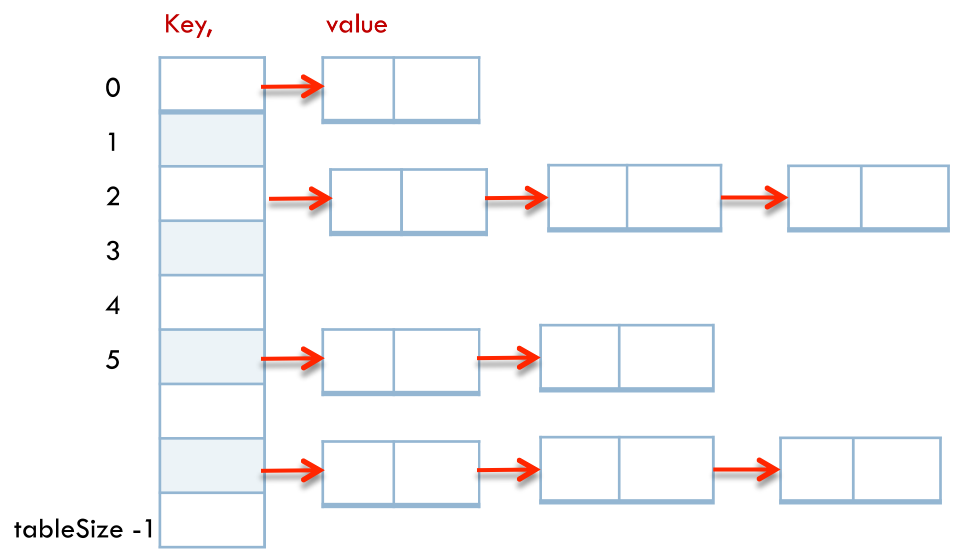
Buckets/chains

- Each entry of the array is an ARRAY (bucket) or a LINKED LIST (chain)

- **Buckets**
  - What is the array size?
  - What is the problem of this design?
    - Too much wasted space

- **Chaining**
  - Each entry is a linked list





Hash Function

- An ideal hash function should map each given key to a unique location in the table
  - Practically, it is unattainable most of the times

- A "good" hash function
  - Simple and fast
  - Scramble data evenly across the array
  - Scatter random keys evenly
    - Table size is 5, keys = [0..14], h(k) = k % 5 does not scatter data randomly
  - Scatter clustered keys evenly

- Rules of thumb
  - Hash function should use the whole key, not a portion of it
  - Whenever use modulo hashing, the base should be prime number

Modulo Arithmetic

- Simple hash function can be $h(k) = k \bmod m$

- If m is NOT a prime. Let m be 10^d or 2^d
  - The keys clustered in the lower order digits will cause collisions
  - For example,
    - $h(k) = k \bmod 100$, k is the birth year

Hash Functions - Numerical Values

- Consider: $h(x) = x\%16$
  - poor distribution, not very random
  - depends solely on least significant four bits of key
- Better, *mid-square* method
  - if keys are integers in range 0,1,…,K , pick integer C such that DC^2 about equal to K^2, then

    $$h(x) = \lfloor x^2/C \rfloor \,\%\, D$$

    extracts middle *r* bits of x^2, where 2^r=D (a base-D digit)
  - better, because most or all of bits of key contribute to result

Relatively Prime Numbers

- Two numbers are relatively prime numbers
  - They do not share any factors other than 1

- If m and n (3 and 7) are relatively prime, their first common multiple is?
  - m * n

- If m (i.e. tableSize) is a prime number (not 2 or 5), what is the first common multiple of 10^d and m?
  - m * 10^d
  - If m = 13 and d = 2, common multiples: 1300, 2600, 3900, etc.

Why Prime?

- Suppose
  - data stored in hash table: 7160, 493, 60, 55, 321, 900, 810
  - tableSize = 10

    data hashes:

    0, 3, <u>0</u>, 5, 1, <u>0</u>, <u>0</u>
  - tableSize = 11

    data hashes to 10, 9, 5, 0, 2, <u>9</u>, 7

Real-life data tends to have a pattern

Being a multiple of 11 is usually *not* the patterns

Table size should be prime, why ?

- Assume a set of numbers $K = \{k_1, k_2, k_3, \ldots\}$
  - They all have the same hash result, which is $k_i \bmod 10^d$
  - For example, 99, 199, 299, 4399

- If tableSize is prime, and we use k mod tableSize to hash
  - What is the chance these numbers hash to the same locations?

Table size should be prime, why ?

- If k_1 and k_2 hash to the same value and table size is 10^d we have
  - $k_1 \bmod 10^d = k_2 \bmod 10^d = r$ (same remainder)

  - We have $k_1 = x * 10^d + r$, $k_2 = y * 10^d + r$,
  - Thus, $k_1 - k_2 = (x - y) * 10^d$

  - In other words, all the keys $k_1 - k_2$ equals to some multiple of 10^d hash to the same value, **collision**!

  - For example, 199 -99 = 100, 299 -99 = 200, 4399 – 299 = 4200

Module hashing

- If the table size is a prime number p instead of 10^d , the hash function map the key to "k mod p".
  - If k_1 and k_2 were to map to the same place, k_1 - k_2 (we know is a power of 10^d) would also have to be divisible by p
  - If k_1 and k_2 to map to the same place, k_1 - k_2 would be some multiple p * 10^d
  - For example, if p = 13 and d =2, k_1- k_2 would have to be 1200, 2600, 3900

Hash Function

- When we build hash table, hash function usually has two steps
  - First, convert the original data to a new value, which should be scattered (not clustered)
    - For example, JAVA use the following function to calculate the hash code for a given string, please note prime number 31 is carefully selected.

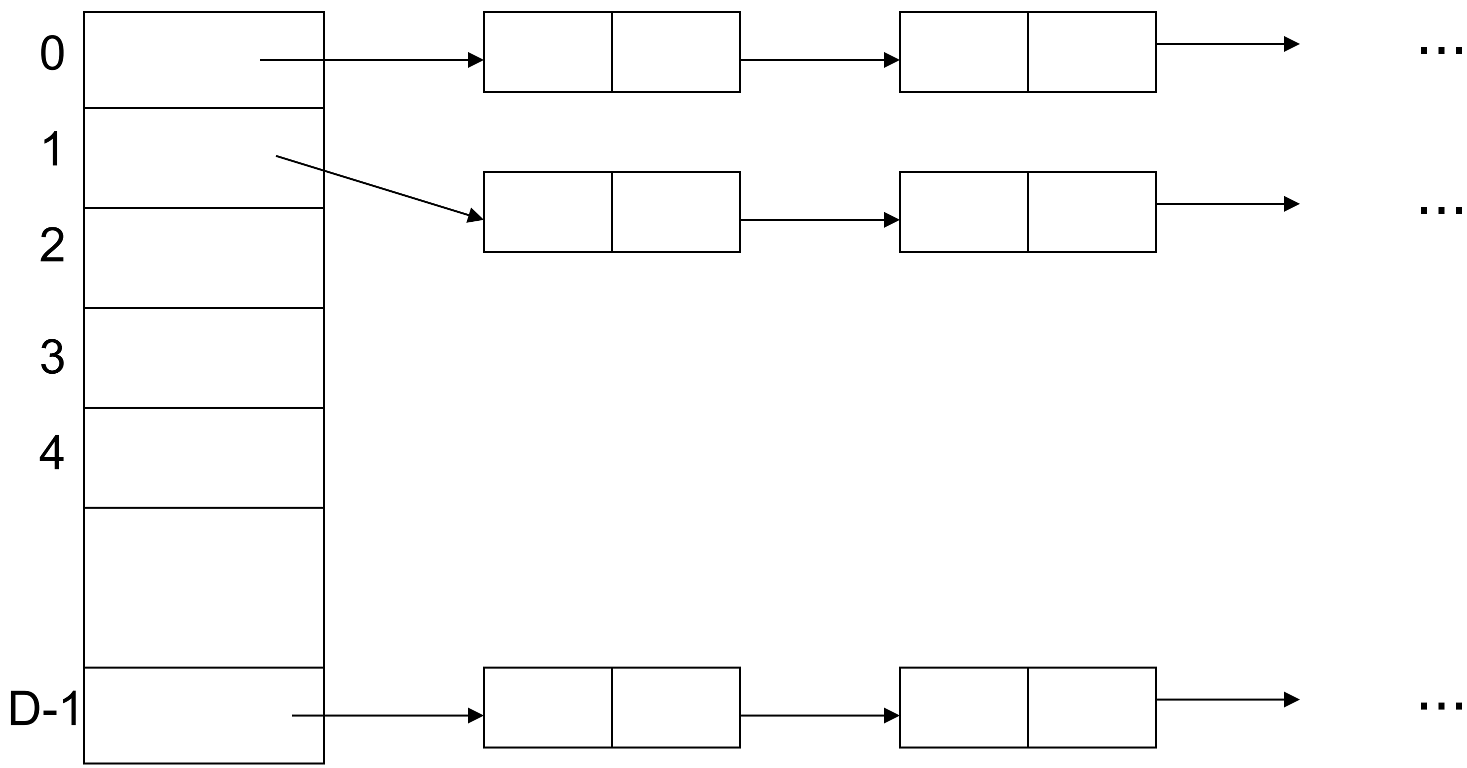
$$h(s) = \sum_{i=0}^{n-1} s[i] \cdot 31^{n-1-i}$$

  - Once the new value is calculated, use a prime number to do module operation, convert it to an index value of array.

Open Hashing

- Each bucket in the hash table is the head of a linked list
- All elements that hash to a particular bucket are placed on that bucket's linked list
- Records within a bucket can be ordered in several ways
  - by order of insertion, by key value order, or by frequency of access order

Open Hashing Data Organization



Analysis

- Open hashing is most appropriate when the hash table is kept in main memory, implemented with a standard in-memory linked list

- We hope that number of elements per bucket roughly equal in size, so that the lists will be short
- If there are n elements in set, then each bucket will have roughly n/D
- If we can estimate n and choose D to be roughly as large, then the average bucket will have only one or two members