OPERATIONS

PURPOSE: Hash tables are used to maintain a (possibly evolving) set of elements, for example: transactions, people + associated data, IP addresses, etc.
* Remember, a set is a grouping of unique elements. That is, no two elements in the set possess equivalency.

Hash tables implement their set property by having elements serve as a key to unlock an element's associated data. For example in a contact list a person's name is the key and the associated data would be the person's e-mail address and phone number. Hash tables can have keys without associated data, but data cannot exist within the hash table without a key.

INSERT: Add a new record
DELETE: Remove an existing record
LOOKUP: Check for a particular record

} All accessed using the key, all $O(1)$

The above operations have constant, $O(1)$, runtime.
<u>IF</u> the hash table data structure is implemented correctly.
<u>IF</u> the data being looked up is non-pathological (more later)

Hash tables do lookups amazingly well. For keeping track of minimums / maximums or order of elements hash tables are not the appropriate data structure.

APPLICATION: DE-DUPLICATION

GIVEN: A stream of objects, think a linear scan through a huge file or objects arriving in real time (Packets sent to a router from various IP addresses).

GOAL: Remove duplicates (Keep track of unique objects)
Examples: Report of unique visitors to a web site
Avoid duplicates when returning search results

SOLUTION:
1. When new object x arrives
2. Lookup x in hash table H
3. If not found, insert x into H
4. Continue until complete and provide H

(con't on next sheet)

APPLICATION: 2-SUM PROBLEM

GIVEN: An unsorted array A of n integers and target value t.

FIND: Whether or not there are two numbers x, y in A such that $x + y = t$

NAIVE SOL'N: $O(n^2)$ — Perform exhaustive search of the array checking the sums of x and y against t.
RUNTIME: $O(n^2)$

We can do better!

BETTER SOL'N: $O(n \lg n)$
1. Sort A - $O(n \lg n)$
2. For each x in A, search for the value $t-x$ in A via binary search - $O(n \lg n)$

We can do better with hash tables!

HASH TABLE SOL'N: $O(n)$
1. Insert elements of A into hash table H. - $O(1)$ n times
2. For each x in A, lookup $t-x$ in H - $O(1)$ n times

OTHER APPLICATIONS:

Blocking network traffic (IP address lookup in blacklist. SPAM!)

Search algorithms - A hash table avoids exploring any configuration (chess pieces on the board) more than once.

Anything needing fast lookups

(con't on next sheet)

IMPLEMENTATION DETAILS

HIGH LEVEL

SETUP: We want to store items in universe U. Examples include: All IP addresses, all names, all chess board configurations, etc.
U is generally pretty big.

GOAL: Want to maintain an evolving set $S \subseteq U$. S is usually a much more reasonable size than U.

NAIVE SOLUTIONS:

ARRAY-BASED SOLUTION (INDEXED BY U)
All elements have a specific place in the array.
PRO: $O(1)$ operation runtime.
CON: $O(U)$ space requirement.

LIST-BASED SOLUTION
Only items in S are utilized and one item points to the next item.
PRO: $O(S)$ space requirement
CON: $O(s)$ operation runtime. (Slowdown from hash table runtime)

HASH TABLE SOLUTION:

1. Pick n. n = number of "buckets", places a value can be stored, with $n \approx$ # of items in S. For simplicity, assume S doesn't vary too much. However, if S did vary enough to cause a need to increase the size of the hash table use rules for resizing arrays. Generally, double the size of the array when you need to increase an array's capacity or halve the size of the array when the array contains 1/4 of its capacity, that is, we only care about 1/4 of an array's total capacity and the other 3/4 are empty or no-longer needed.

2. Choose a hash function h such that given an element x of U it fits in (gets assigned to) one of the n buckets.
$h: U \rightarrow \{\emptyset, 1, 2, \ldots, n-1\}$

3 Using an array of length n, store the element x in $A[h(x)]$.

(con't on next sheet)

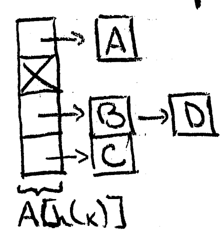
IMPLEMENTATION DETAILS (con't)

RESOLVING COLLISIONS

Collisions occur when a hashing function assigns the same value (puts an element in the same "bucket") to two or more different elements. This happens quite frequently even with large datasets and must be addressed. Hoping there are no collisions is not a solution.

COLLISION: Distinct x and y in universe U, such that $h(x) = h(y)$

SOLUTION #1: CHAINING / SEPERATE CHAINING (EASIER DELETION OPS)
Keep a linked list in each bucket
Given a key/object x perform insert/delete/lookup in the list in $\underbrace{A[\underbrace{h(x)}_{\text{BUCKET ASSIGNMENT FOR X}}]}_{\text{LINKED LIST FOR X}}$



SOLUTION #2: OPEN ADDRESSING (USEFUL IF SPACE IS AT A PREMIUM)
No linked lists allowed. Only one object allowed per bucket.
Hash function now specifies a probe sequence (keep trying hash functions until you find an open slot) $h_1(x), h_2(x), \ldots$

EXAMPLES:
LINEAR PROBING: Perform a hash function to determine which bucket x should go in, then continue looking in consecutive slots until an open slot occurs

DOUBLE HASHING: Have two different hash functions h_1, h_2
1. Run $h_1(x)$ and check if the space is open
If it's open put the object in the bucket.
If not run h_2
Add the value from h_2 to the value to h_1
Check the $h_1 + h_2$ slot
If it's open put the object in the bucket
If not add h_2 to the previous value
Continue until an open slot is found.

IMPLEMENTATION DETAILS (con't)

MAKING A GOOD HASH FUNCTION

WHY A GOOD HASH FUNCTION MATTERS

Say we have a hash table utilizing chaining to resolve collision issues. Using chaining the insert operation is constant, O(1), runtime. This is because if there is a case where two objects are going to be placed in the same bucket, the newest object is inserted at the front of the list.

However, since the buckets now contain a list of objects in the worst case for a lookup or deletion operation we will have to traverse the whole list in the bucket so the runtime will be O(list length). A really poor choice for a hash function can drop all the elements into one bucket leading to O(n) runtime for lookups and deletions completely negating the constant time benefits of a hash table.

Open addressing can also suffer from the same problem of possible linear runtime during its probe sequence (insertion). Poor hash functions would probe the same spots every time an insertion occurs rather than spreading the data into the available buckets fairly evenly.

Therefore, performance of the hash table directly depends on the performance of the hash function.

PROPERTIES OF A GOOD HASH FUNCTION

1. Should lead to good performance by spreading data fairly evenly throughout all possible buckets.
   Chaining implementation has lists of equal length.
   Open addressing probes about the same number of times for each insertion.
   The gold standard is completely random hashing.

2. The hash function should be easy to store information about the result of putting an object through the hash function.
   The hash function should be very fast (O(1)) to evaluate.
   If a hash function takes longer than constant time to run it completely negates the constant time performance of hash table operations.

(con't on next sheet)

3-0235 — 50 SHEETS — 5 SQUARES
3-0236 — 100 SHEETS — 5 SQUARES
3-0237 — 200 SHEETS — 5 SQUARE
3-0137 — 200 SHEETS — FILLER

COMET

IMPLEMENTATION DETAILS (cont)

MAKING A GOOD HASH FUNCTION (cont)

WHAT NOT TO DO

EXAMPLE: KEYS = 10-DIGIT PHONE NUMBERS.
Possible combinations in the universe $U = 10^{10}$

We are only interested in about 500 of the numbers in the universe. When making our choice for n to store the set of 500, S, let's make n double S to ensure we don't need to increase or decrease the number of buckets needed.

$n = 10^3$

TERRIBLE IMPLEMENTATION:
Let x represent the area code (first 3 digits) of the phone number. So the bucket a number is placed in will be decided by h(x) using chaining.

This is terrible for two reasons:
1. The area code can represent a lot of people, 10^7. Therefore, you could have a bucket representing an area code with a everyone in the set in one single bucket leading to very slow lookup times.
2. Some area codes are not even utilized leaving some buckets of n completely empty while causing further clustering with the hash function. Not distributed randomly.

MEDIOCRE IMPLEMENTATION
Let x represent the last 3 digits of the phone number.

This assumes that the last 3 digits of phone numbers are uniformly distributed which has no evidence for being true.

(cont on next sheet)

IMPLEMENTATION DETAILS (cont)

MAKING A GOOD HASH FUNCTION (cont)

WHAT NOT TO DO (cont)

EXAMPLE: KEYS = COMPUTER MEMORY LOCATIONS

Memory is allocated in bytes - 4 binary digits (multiples of 4)

So, memory locations will be multiples of a power of 2 and also even.

We'll still want to work with 500 objects so $n = 1000$.

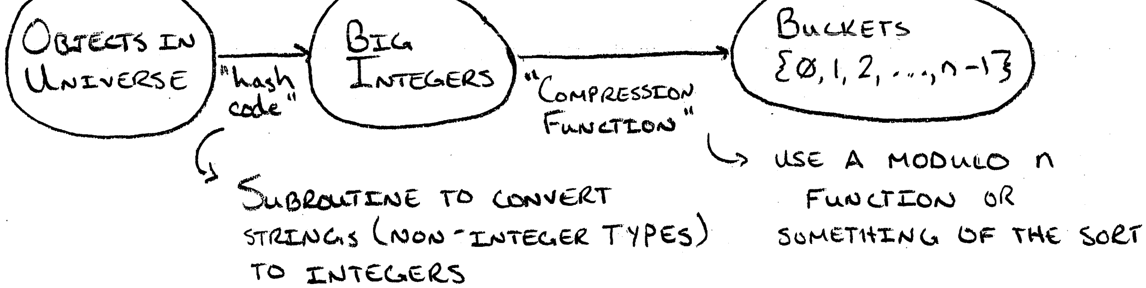
BAD HASH FUNCTION: Since we have 1000 buckets let's just see how the least significant byte is divided by 1000 and let the remainder of the modulo operation be the bucket to store the memory location.

$h(x) = x \% 1000$

This is bad because as stated above all the memory locations will be placed into even buckets and all the odd buckets are guaranteed to be empty. The hash function cannot provide an odd number.

QUICK AND DIRTY HASH FUNCTIONS

These ideas following are good for prototyping or quick to code up.
Do not use for production environments. Go and learn better hash functions for the problem domain.



HOW TO CHOOSE n, # OF BUCKETS

1. Choose n to be a prime number (within constant factor of objects in the table).
2. Not too close to a power of 2 or 10.

IMPLEMENTATION DETAILS (con't)

THE LOAD OF A HASH TABLE

The load factor, α, of a hash table is

$$\alpha = \frac{\text{\# of objects in hash table}}{\text{\# of buckets in hash table, } n}$$

α must be a constant, $\alpha = O(1)$, for operations to run in constant time. That is you want α to be less than and not close to 1, especially for open addressing implementations.

For example, using a hash table implemented with chaining if you have n buckets and $n \lg n$ objects $\alpha > 1$. Because all objects will fit in the buckets due to the chaining each bucket, on average, will have $\lg n$ elements. Now the lookup operation goes from constant time, $O(1)$, to logarithmic time, $O(\lg n)$, due to having to traverse the lists in the hash table. See page 5, WHY A GOOD HASH FUNCTION MATTERS, for a refresher.

Therefore, for good hash table performance a strong implementation will grow and shrink the number of buckets algorithmically to control α.

PATHOLOGICAL DATASETS

We know that for constant time hash table performance we need a good hash function. Ideally, a clever hash function that spreads the dataset evenly across the buckets.

Unfortunately, there is not a universal hash function which will perform ideally or near ideally in every case. For every hash function there is a pathological dataset that will make even the best, most clever hash function perform in a worst-case, $O(n)$, manner. This is due to the compression from the hash function. Essentially a bad actor selects elements for a data set so that when hashed all go into the same bucket leading to worst-case performance.

(con't on next sheet)

IMPLEMENTATION DETAILS (cont)

PATHOLOGICAL DATASETS (cont)

SOLUTIONS

1. Use a cryptographic hash function, for example, SHA-2. This is a potential solution because unlike a simple hash function with a cryptographic hash function it is considered infeasible to reverse engineer a pathological dataset when implemented correctly.
2. Use randomization. That is, design a family of hash functions, H, such that for all datasets S, "almost all" of the hash functions in H spread S out fairly evenly. Think of the randomization used to select a pivot value in quicksort, but applied to hash functions.

UNIVERSAL HASH FUNCTION FAMILIES

For a hash function to be universal it must be capable of spreading elements of a set fairly uniformly across the n buckets used to hold the set. The gold standard is a function that performs similarly to a uniform random distribution. Since we have determined every hash function has a pathological dataset no matter how well designed the hash function is there must be a way to mitigate the determined person attempting to subvert our work or the unknowing user who happens to get very unlucky with their dataset. This leads to hash function families.

DEFINITION: Let H be a set of hash functions for any dataset universe U from $\{0, 1, 2, \ldots, n-1\}$

H is universal if and only if:

1. for all $x, y \in U$ with $x \neq y$
2. $$\Pr_{h \in H}\left[\begin{matrix} x, y \text{ collide} \\ h(x) = h(y) \end{matrix}\right] \leq \frac{1}{n}, \quad n = \#\text{ of buckets}$$

when h is chosen uniformly at random from H.

This leads to a collision probability as small as the gold standard of perfectly random hashing.

(cont on next sheet)

IMPLEMENTATION DETAILS (con't)

UNIVERAL HASH FUNCTION FAMILIES (con't)

EXAMPLE: HASHING IP ADDRESSES

Let U = IP addresses, an IP address is a 32 bit integer with 4 different 8 bit parts. So, the parts can range in value from $\emptyset$ to 255, 2^8, giving possible IP addresses of (0.0.0.0) to (255.255.255.255). As such we can represent each part using the form (x_1, x_2, x_3, x_4) with each $x_i \in \{\emptyset, 1, 2, \ldots, 255\}$.

Say we're interested in being able to look up and access a set of about 500 IP addresses. From the QUICK AND DIRTY HASH FUNCTIONS section we want to select a number of buckets, n, sufficient to store the set.

Let n = a prime number not too close to a power of 2 or 10 and greater than the maximum value set for a_i below.

Let $n = 977$.

CONSTRUCTION OF THE HASH FAMILY

Define one hash function, h_a, per four tuple, $a = (a_1, a_2, a_3, a_4)$ with each $a_i \in \{\emptyset, 1, 2, \ldots, n-1\}$. Therefore, a can range from $(\emptyset, \emptyset, \emptyset, \emptyset)$ to (976, 976, 976, 976) and any combination of values in between. With this definition of a we can see that there are $n \cdot n \cdot n \cdot n$ (n^4) different combinations possible for a.

A hash function yields a single bucket number. To get from a 4-part IP address to a single bucket number we can define the hash function, h_a, as:

$$h_a(x) = (\underset{\text{DOT PRODUCT}}{a \cdot} x^{T \leftarrow \text{TRANSPOSE}}) \% n = \left([a_1 \; a_2 \; a_3 \; a_4] \cdot \begin{bmatrix} x_1 \\ x_2 \\ x_3 \\ x_4 \end{bmatrix} \right) \% n \rightarrow$$

$$h_a(x) = \underbrace{(a_1 x_1 + a_2 x_2 + a_3 x_3 + a_4 x_4) \% n}_{\text{YIELDS A BUCKET \# FROM } \emptyset \text{ TO } n-1}$$

Leads to constant time to evaluate and constant time to store the hash value in a bucket leading to a hash function family of n^4 hash functions when a_i is selected uniformly at random.

COMET

3-0235 — 50 SHEETS — 5 SQUARES
3-0236 — 100 SHEETS — 5 SQUARES
3-0237 — 200 SHEETS — 5 SQUARES
3-0137 — 200 SHEETS — FILLER

IMPLEMENTATION DETAILS (con't)

UNIVERSAL HASH FUNCTION FAMILIES (con't)

EXAMPLE: HASHING IP ADDRESSES (con't)

From the previous page the universal family of hash functions is now defined:

$$H = \{ h_a \mid a_1, a_2, a_3, a_4 \in \{\emptyset, 1, 2, \ldots, n-1\} \}$$

PROOF THIS HASH FAMILY IS UNIVERSAL

Consider distinct IP addresses (x_1, x_2, x_3, x_4), (y_1, y_2, y_3, y_4)

Assume x and y are the same except $x_4 \neq y_4$

We need to prove that the possibility $h(x) = h(y)$ is at most $\frac{1}{n}$, or what fraction of the hash functions in the family of hash functions will cause a collision.

$$\Pr_{h_a \in H} [h_a(x_1, x_2, x_3, x_4) = h_a(y_1, y_2, y_3, y_4)]$$

collision occurs when

$$(a_1x_1 + a_2x_2 + a_3x_3 + a_4x_4)\%n = (a_1y_1 + a_2y_2 + a_3y_3 + a_4y_4)\%n$$

Using principle of deferred decisions (sometimes fixing some of the random inputs clarifies the role the remaining randomness / the effect of the remaining randomness on the problem.)

So, suppose we have already determined a_1, a_2, and a_3.

collision occurs when $a_4(x_4 - y_4)\%n = \underbrace{\sum_{i=1}^{3} a_i(y_i - x_i)\%n}_{\text{some fixed \# in } \{\emptyset, 1, 2, \ldots, n-1\}}$

Now, how many choices of a_4 will cause a collision?

(con't on next sheet)

IMPLEMENTATION DETAILS (con't)

UNIVERSAL HASH FUNCTION: FAMILIES (con't)

EXAMPLE: HASHING IP ADDRESSES (con't)

PROOF THIS HASH FAMILY IS UNIVERSAL (con't)

Since $x_4 \neq y_4$, $x_4 - y_4 \neq \emptyset$

n is prime

a_4 is selected uniformly at random (equal chance to have $a_4 = \emptyset$ or $a_4 = n-1$ or any value in between.)

Proof with a small prime n that the result of $a_4(x_4-y_4)\%n$ is also uniform at random.

Let $n=7$, $x_4 - y_4 = 2$

$a_4 = \emptyset \rightarrow a_4(x_4-y_4)\%n = \emptyset(2)\%7 = \emptyset$

$a_4 = 1 \rightarrow 1(2)\%7 = 2$

$a_4 = 2 \rightarrow 2(2)\%7 = 4$

$a_4 = 3 \rightarrow 3(2)\%7 = 6$

$a_4 = 4 \rightarrow 4(2)\%7 = 1$

$a_4 = 5 \rightarrow 5(2)\%7 = 3$

$a_4 = 6 \rightarrow 6(2)\%7 = 5$

All values of a_4 from $\emptyset$ to $n-1$ are equally likely to provide a value from $\emptyset$ to $n-1$ given the above conditions.

Using the proof with a small prime and the fact that a collision occurs when $a_4(x_4-y_4)\%n = \sum_{i=1}^{3}(y_i - x_i)\%n$ there can only be one point in the range of n which satisfies the above equations since all outcomes are equally likely.

Therefore, $\Pr_{h_a \in H}[h_a(x) = h_a(y)] = 1/n$.