Hashtables

CS 5002: Discrete Math and Data Structures

Adrienne Slaughter

Northeastern University

Apr 2, 2018

Outline

1. Why hashtables?
2. Hash functions
3. Uses for Hash tables
4. Collisions
5. Performance
6. Load Factor
7. Implementation

How much is this onion?

Ways to answer this question:

How much is this onion?

Ways to answer this question:

- Look it up in a sorted list:

How much is this onion?

Ways to answer this question:

- Look it up in a sorted list: $O(\log n)$

How much is this onion?

Ways to answer this question:

- Look it up in a sorted list: $O(\log n)$
- Look it up in an unsorted list:

How much is this onion?

Ways to answer this question:

- Look it up in a sorted list: $O(\log n)$
- Look it up in an unsorted list: $O(n)$

How much is this onion?

Ways to answer this question:

- Look it up in a sorted list: $O(\log n)$
- Look it up in an unsorted list: $O(n)$
- Ask your buddy if she remembers:

How much is this onion?

Ways to answer this question:

- Look it up in a sorted list: $O(\log n)$
- Look it up in an unsorted list: $O(n)$
- Ask your buddy if she remembers: $O(1)$

Comparing Runtimes

Num Items	Simple Search $O(n)$[1]	Binary Search $O(\log n)$	Buddy $O(1)$

[1] assume 1/10 second per comparison

Comparing Runtimes

Num Items	Simple Search $O(n)$[1]	Binary Search $O(\log n)$	Buddy $O(1)$
100	10 sec	1 sec	Instant

[1] assume 1/10 second per comparison

Comparing Runtimes

Num Items	Simple Search $O(n)$[1]	Binary Search $O(\log n)$	Buddy $O(1)$
100	10 sec	1 sec	Instant
1000	1.6 min	1 sec	Instant

[1] assume 1/10 second per comparison

Comparing Runtimes

Num Items	Simple Search $O(n)$[1]	Binary Search $O(\log n)$	Buddy $O(1)$
100	10 sec	1 sec	Instant
1000	1.6 min	1 sec	Instant
10000	16.6 min	2 sec	Instant

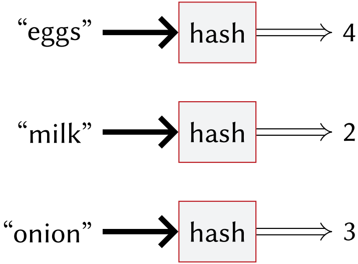
[1] assume 1/10 second per comparison

How can we replicate Buddy with a data structure?

(eggs, \$2.49)	(milk, \$4.29)	(onion, \$0.29)

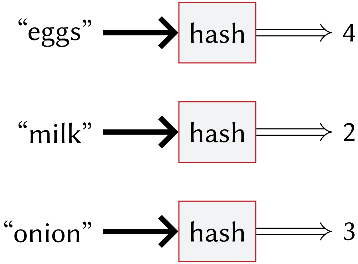
- Each item is essentially 2 items: a product name, and a price
- If we sort by name, we can find a product in $O(\log n)$ time
- How can we get down to $O(1)$?

Hash Function



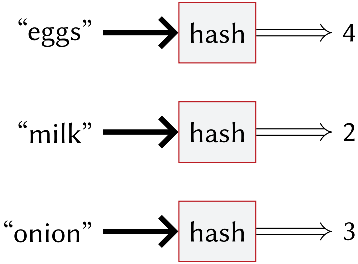
- Transforms a {string, number, struct, object, ...} to a number

Hash Function



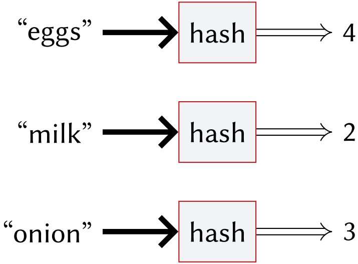
- Transforms a {string, number, struct, object, ... } to a number
- Must be consistent. Every time you give it the same input, it returns the same value.

Hash Function



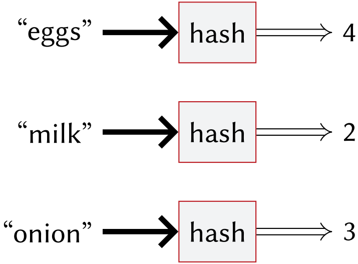
- Transforms a {string, number, struct, object, …} to a number
- Must be consistent. Every time you give it the same input, it returns the same value.
    - Every time you give it "eggs", it returns '4'.

Hash Function



- Transforms a {string, number, struct, object, ...} to a number
- Must be consistent. Every time you give it the same input, it returns the same value.
  - Every time you give it “eggs”, it returns ‘4’.
- Must be distinguishing. In the best case, it returns a different value for every distinct input given to it.

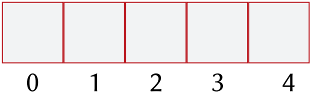
Hash Function



- Transforms a {string, number, struct, object, ...} to a number
- Must be consistent. Every time you give it the same input, it returns the same value.
  - Every time you give it “eggs”, it returns ‘4’.
- Must be distinguishing. In the best case, it returns a different value for every distinct input given to it.
  - A function that always returns 1 is not helpful.

How can we use the hash function to make our “Buddy”?

If we have an array:

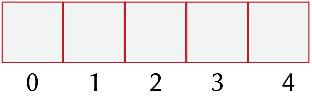


we can use the hash function to generate an index to the array.

Map a given name (“eggs”) to a number (4), and use that number as the index of where to store that value.

How can we use the hash function to make our “Buddy”?

If we have an array:



we can use the hash function to generate an index to the array.

Map a given name (“eggs”) to a number (4), and use that number as the index of where to store that value.

How can we use the hash function to make our "Buddy"?

If we have an array:

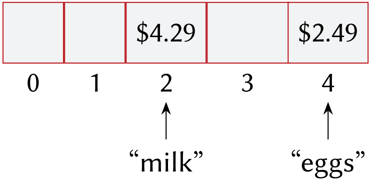
0	1	2	3	4
				\$2.49

we can use the hash function to generate an index to the array.

Map a given name ("eggs") to a number (4), and use that number as the index of where to store that value.

How can we use the hash function to make our “Buddy”?

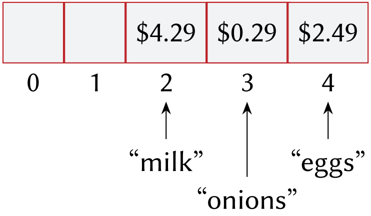
Add milk to the array, using the hash function...



How can we use the hash function to make our “Buddy”?

Add milk to the array, using the hash function…

…and then add onions:



Retrieving data using the hash function

Now, when we need to look up the cost of an item:

- Use the hash function to get the array index for the given product

Retrieving data using the hash function

Now, when we need to look up the cost of an item:

- Use the hash function to get the array index for the given product
- Return the price stored in the array at that location

Retrieving data using the hash function

Now, when we need to look up the cost of an item:

- Use the hash function to get the array index for the given product
- Return the price stored in the array at that location

$\Rightarrow$ we now have a Buddy!

Retrieving data using the hash function

Now, when we need to look up the cost of an item:

- Use the hash function to get the array index for the given product
- Return the price stored in the array at that location

$\Rightarrow$ we now have a Buddy!

- The hash function always returns the same value for the same input

Retrieving data using the hash function

Now, when we need to look up the cost of an item:

- Use the hash function to get the array index for the given product
- Return the price stored in the array at that location

$\Rightarrow$ we now have a Buddy!

- The hash function always returns the same value for the same input
- The hash function returns a different value for different inputs

Retrieving data using the hash function

Now, when we need to look up the cost of an item:

- Use the hash function to get the array index for the given product
- Return the price stored in the array at that location

$\Rightarrow$ we now have a Buddy!

- The hash function always returns the same value for the same input
- The hash function returns a different value for different inputs
- The hash function knows how big the array is, so always returns something within those bounds.

New Data Structure: Hash Table

A ***hash table*** is a combination of a hash function and array.

Other names:

- hash map
- map
- dictionary
- associative array

Which of these hash functions are consistent?

The hash function must be consistent about returning the same output for the same input.

If it doesn't, you won't be able to retrieve the data you store in the hash table.

Which of these is consistent?

1. $f(x) = 1$
2. $f(x) =$rand()
3. $f(x) =$next_empty_slot()
4. $f(x) =$len(x)

Use Cases

- Look up tables
  - Phone book. key = name, value = phone number
  - Websites. key = URL, value = IP address

Use Cases

- Look up tables
  - Phone book. key = name, value = phone number
  - Websites. key = URL, value = IP address
- Prevent duplicate entries
  - Voting roster: ensure the same person doesn't vote twice

Use Cases

- Look up tables
  - Phone book. key = name, value = phone number
  - Websites. key = URL, value = IP address
- Prevent duplicate entries
  - Voting roster: ensure the same person doesn't vote twice
- As a cache
  - Store a copy of a web page locally rather than requesting the entire thing over the network
  - key = URL, value = webpage

Hash table operations and vocab

A few hash table-specific terms:

- ***key***
- ***value***
- ***put***: `put(hashtable, key, value)`
- ***get***: `get(hashtable, key)` (returns the value)

Hash Table: `get()` and `put()`

HASH-INSERT(T, key, val)

T[h(key)] = val

HASH-SEARCH(T, key)

return T[h(key)]

Simple Hash Table implementation

```c
#include<stdio.h>
#include<strings.h>

int table[20] = {0};

int hash(char *key){
  return strlen(key);
}

void put(char *key, int value){
  table[hash(key)] = value;
}

int get(char *key){
  return table[hash(key)];
}

int main(){
  put("onion", 3);
  put("tomato", 4);
  put("red pepper", 15);

  printf("value for onion: %d\n", get("onion"));
  printf("value for onion: %d\n", get("tomato"));
  printf("value for onion: %d\n", get("red pepper"));
}

```

Listing 1: hash table implementation

Demo...

Collisions

So far, our understanding of hash tables is predicated on the ***hash function***.

Specifically, that the has function generates a **unique** value for every input.

Collisions

So far, our understanding of hash tables is predicated on the ***hash function***.

Specifically, that the has function generates a **unique** value for every input.

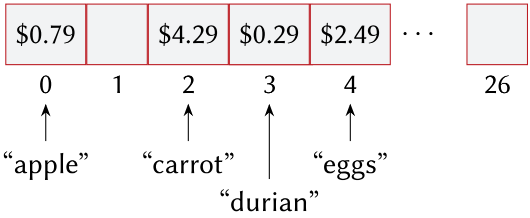
Let's say our function returns the position in the alphabet of the first letter of the input.

Collisions

So far, our understanding of hash tables is predicated on the ***hash function***.

Specifically, that the has function generates a **unique** value for every input.

Let's say our function returns the position in the alphabet of the first letter of the input.

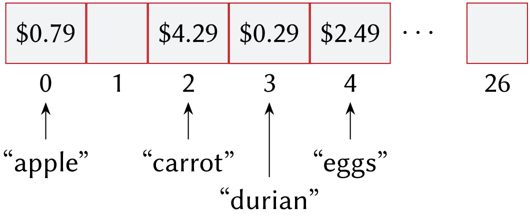


Collisions

So far, our understanding of hash tables is predicated on the ***hash function***.

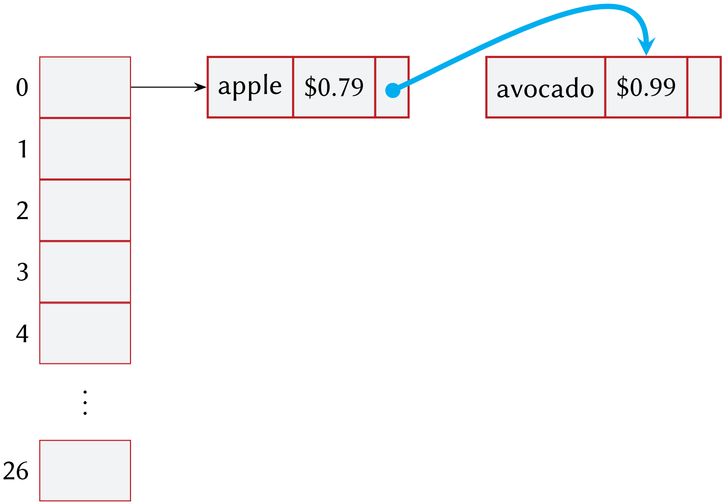
Specifically, that the has function generates a **unique** value for every input.

Let's say our function returns the position in the alphabet of the first letter of the input.



What happens when I go to add "avocado"?

Collisions: One Solution is Chaining



Next time: add slide on probability of collisions. Birthday paradox, Pigeonhole principle

Hash Table: `get()` and `put()` with chaining

HASH-INSERT(T, key, val)

1 insert val at head of list T[h(key)]

HASH-SEARCH(T, key)

1 search for element with key in list T[h(key)]

Hashtable Performance

	Average Case	Worst case
Search (get)	O(1)	
Insert (put)	O(1)	
Delete	O(1)	

In the Average Case, we get a hit right away— don't have to search through the list.

This is equivalent to our Buddy saying "Apples are $0.29".

Hashtable Performance

	Average Case	Worst case
Search (get)	O(1)	O(n)
Insert (put)	O(1)	O(1)
Delete	O(1)	O(n)

In the Average Case, we get a hit right away— don't have to search through the list.
This is equivalent to our Buddy saying "Apples are $0.29".

In the Worst Case, we need to search through all the items in a chained linked list.

This is when our Buddy says "I don't know– look in the book".

What does our hash table look like...?

Add items: avocado, apple, asparagus, ...

$\Rightarrow$ Your hash function is super important!! (You want one that will map keys evenly)

A good hash function keeps those linked lists small too.

Load factor

$$\frac{\text{number of items in hash table}}{\text{total number of slots}}$$

Another way of saying it: The average number of items in a chain

Load factor

$$\frac{\text{number of items in hash table}}{\text{total number of slots}}$$

Another way of saying it: The average number of items in a chain What's the load factor for this hash table?

		\$4.29	\$0.29	\$2.49
0	1	2	3	4

Load factor

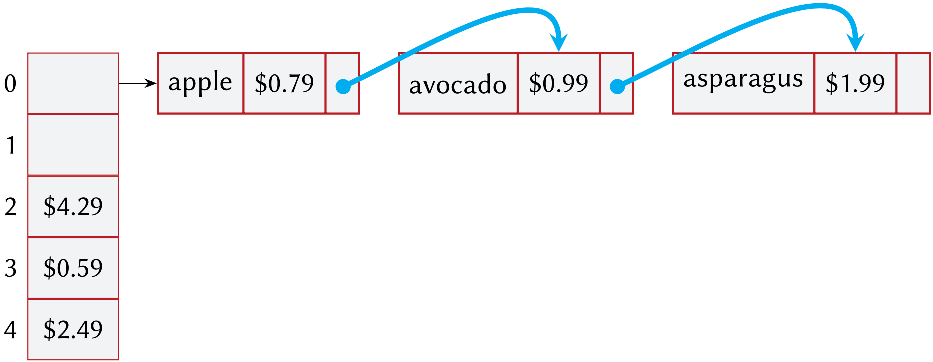
$$\frac{\text{number of items in hash table}}{\text{total number of slots}}$$

Another way of saying it: The average number of items in a chain What's the load factor for this hash table?

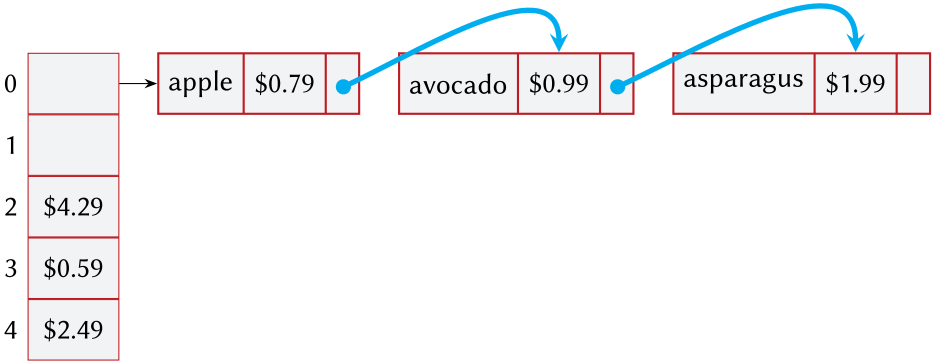
		\$4.29	\$0.29	\$2.49
0	1	2	3	4

3/5

What's the load factor for this table?



What's the load factor for this table?



6/5

Load Factor

Load Factor

- If your hash table has 50 slots and 50 items, the load factor is 1.

Load Factor

- If your hash table has 50 slots and 50 items, the load factor is 1.
- If your hash table has 25 slots and 50 items, the load factor is 2.

Load Factor

- If your hash table has 50 slots and 50 items, the load factor is 1.
- If your hash table has 25 slots and 50 items, the load factor is 2.
  - There are, on average, 2 items per slot.

Load Factor

- If your hash table has 50 slots and 50 items, the load factor is 1.
- If your hash table has 25 slots and 50 items, the load factor is 2.
  - There are, on average, 2 items per slot.
  - This means we **can NOT** take advantage of the best case scenario.

Load Factor

- If your hash table has 50 slots and 50 items, the load factor is 1.
- If your hash table has 25 slots and 50 items, the load factor is 2.
  - There are, on average, 2 items per slot.
  - This means we **can NOT** take advantage of the best case scenario.
- The lower the load factor, the lower the chance of collisions.

Load Factor

- If your hash table has 50 slots and 50 items, the load factor is 1.
- If your hash table has 25 slots and 50 items, the load factor is 2.
  - There are, on average, 2 items per slot.
  - This means we **can NOT** take advantage of the best case scenario.
- The lower the load factor, the lower the chance of collisions.
- The higher the load factor, the higher the chance of collisions.

Load Factor

- If your hash table has 50 slots and 50 items, the load factor is 1.
- If your hash table has 25 slots and 50 items, the load factor is 2.
  - There are, on average, 2 items per slot.
  - This means we **can NOT** take advantage of the best case scenario.
- The lower the load factor, the lower the chance of collisions.
- The higher the load factor, the higher the chance of collisions.
- How can we deal with this?

Resizing the table

Resizing the table

- Improve the load factor by resizing the table!

Resizing the table

- Improve the load factor by resizing the table!
- Create a new array, twice as big as the original

Resizing the table

- Improve the load factor by resizing the table!
- Create a new array, twice as big as the original
- Copy all the elements from the original to the new

Resizing the table

- Improve the load factor by resizing the table!
- Create a new array, twice as big as the original
- Copy all the elements from the original to the new
- Load factor improved! Now, a lower chance of collisions

Resizing the table

- Improve the load factor by resizing the table!
- Create a new array, twice as big as the original
- Copy all the elements from the original to the new
- Load factor improved! Now, a lower chance of collisions
- Resizing is expensive (in time), but it averages out overall by allowing us an improved access time

Back to the hash function...

A good hash function give us:

		$4.29		$2.49
0	1	2	3	4

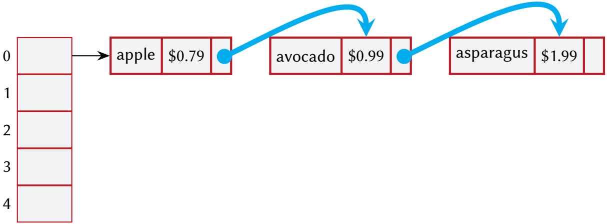
(Items distributed, one per bucket)

Back to the hash function...

A good hash function give us:

(Items distributed, one per bucket)

A bad hash function gives us:



Hash Functions

What makes a good hash function?

[2] radix-128 integer

Hash Functions

What makes a good hash function?

- Each key is equally likely to hash to any of m slots

[2] radix-128 integer

Hash Functions

What makes a good hash function?

- Each key is equally likely to hash to any of m slots
- A common approach is to derive the hash value in a way that is expected to be different than any patterns that exist in the data/keys

[2] radix-128 integer

Hash Functions

What makes a good hash function?

- Each key is equally likely to hash to any of m slots
- A common approach is to derive the hash value in a way that is expected to be different than any patterns that exist in the data/keys
  - Example: The remainder of the key divided by a prime number

[2] radix-128 integer

Hash Functions

What makes a good hash function?

- Each key is equally likely to hash to any of m slots
- A common approach is to derive the hash value in a way that is expected to be different than any patterns that exist in the data/keys
  - Example: The remainder of the key divided by a prime number
- Going from a string to a number:

[2] radix-128 integer

Hash Functions

What makes a good hash function?

- Each key is equally likely to hash to any of m slots
- A common approach is to derive the hash value in a way that is expected to be different than any patterns that exist in the data/keys
  - Example: The remainder of the key divided by a prime number
- Going from a string to a number:
  - `pt` $\Rightarrow$ (112, 116) (ASCII vals) $\Rightarrow (112 \cdot 128) + 116 = 14452$[2]

[2] radix-128 integer

Division Method

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

- m is the size of the table
- k is the key
- Usually pretty fast
- Want to avoid certain values of m:
  - Powers of 2: if $m = 2^p$, $h(k)$ is the p-lowest order bits of k
  - Powers of 10: if keys are decimal numbers
  - $m = 2^p - 1$ when k is a string interpreted in radix-2^p
- General good values of m:
  - Primes not too close to powers of 2

Multiplication Method

$$h(k) = \lfloor m(kA \mod 1) \rfloor$$

- $kA \mod 1$ is the fractional part of kA (or $kA - \lfloor kA \rfloor$)
- Nice becaues the value of m is not critical
- m is typically a power of 2

Universal Hashing

$$h(k) = \lfloor m(kA \mod 1) \rfloor$$

Universal Hashing

$$h(k) = \lfloor m(kA \mod 1) \rfloor$$

- Any fixed hash function is vulnerable to someone choosing the keys such that they always evaluate to the same value.

Universal Hashing

$$h(k) = \lfloor m(kA \mod 1) \rfloor$$

- Any fixed hash function is vulnerable to someone choosing the keys such that they always evaluate to the same value.
- Would make our hash table perform in worst time all the time

Universal Hashing

$$h(k) = \lfloor m(kA \mod 1) \rfloor$$

- Any fixed hash function is vulnerable to someone choosing the keys such that they always evaluate to the same value.
- Would make our hash table perform in worst time all the time
- Solution: choose the hash function **randomly**, *independent* of the keys that will be stored.

Universal Hashing

$$h(k) = \lfloor m(kA \mod 1) \rfloor$$

- Any fixed hash function is vulnerable to someone choosing the keys such that they always evaluate to the same value.
- Would make our hash table perform in worst time all the time
- Solution: choose the hash function **randomly**, *independent* of the keys that will be stored.
- Results in good performance on the average.

Universal Hashing

$$h(k) = \lfloor m(kA \mod 1) \rfloor$$

- Any fixed hash function is vulnerable to someone choosing the keys such that they always evaluate to the same value.
- Would make our hash table perform in worst time all the time
- Solution: choose the hash function **randomly**, *independent* of the keys that will be stored.
- Results in good performance on the average.
- Select the hash function at random at run time from a *carefully designed* class of functions.

Universal Hashing

$$h(k) = \lfloor m(kA \mod 1) \rfloor$$

- Any fixed hash function is vulnerable to someone choosing the keys such that they always evaluate to the same value.
- Would make our hash table perform in worst time all the time
- Solution: choose the hash function **randomly**, *independent* of the keys that will be stored.
- Results in good performance on the average.
- Select the hash function at random at run time from a *carefully designed* class of functions.
  - This randomization means the algorithm runs differently on each execution, on the same input, and guarantees good average-case performance.

Universal Hashing

$$h(k) = \lfloor m(kA \mod 1) \rfloor$$

- Any fixed hash function is vulnerable to someone choosing the keys such that they always evaluate to the same value.
- Would make our hash table perform in worst time all the time
- Solution: choose the hash function **randomly**, *independent* of the keys that will be stored.
- Results in good performance on the average.
- Select the hash function at random at run time from a *carefully designed* class of functions.
  - This randomization means the algorithm runs differently on each execution, on the same input, and guarantees good average-case performance.
  - The collection of hash functions map a given universe of keys into the range $\{0, 1, \ldots, m-1\}$.

Universal Hashing

$$h(k) = \lfloor m(kA \mod 1) \rfloor$$

- Any fixed hash function is vulnerable to someone choosing the keys such that they always evaluate to the same value.
- Would make our hash table perform in worst time all the time
- Solution: choose the hash function **randomly**, *independent* of the keys that will be stored.
- Results in good performance on the average.
- Select the hash function at random at run time from a *carefully designed* class of functions.
  - This randomization means the algorithm runs differently on each execution, on the same input, and guarantees good average-case performance.
  - The collection of hash functions map a given universe of keys into the range $\{0, 1, \ldots, m-1\}$.
  - A hash function randomly chosen from H provides a chance of collision between x and y when $x \neq y$ of $1/m$.

Universal Hashing

$$h(k) = \lfloor m(kA \mod 1) \rfloor$$

- Any fixed hash function is vulnerable to someone choosing the keys such that they always evaluate to the same value.
- Would make our hash table perform in worst time all the time
- Solution: choose the hash function **randomly**, *independent* of the keys that will be stored.
- Results in good performance on the average.
- Select the hash function at random at run time from a *carefully designed* class of functions.
  - This randomization means the algorithm runs differently on each execution, on the same input, and guarantees good average-case performance.
  - The collection of hash functions map a given universe of keys into the range $\{0, 1, \ldots, m-1\}$.
  - A hash function randomly chosen from H provides a chance of collision between x and y when $x \neq y$ of $1/m$.

Designing a universal class of hash functions

- m: table size, make it prime
- key x is $r + 1$ bytes: $x = \{x_0, x_1, \ldots x_r\}$
- $a = \{a_0, a_1, \ldots a_r\}$ is a sequence elements chosen randomly from the set of indices in m
- a

$$h_a(x) = \sum_{i=0}^{r} a_i x_i \mod m$$

Hash Table: Storage

- If the universe $|U|$ is large, ...
- When the set of K keys stored is much smaller than the universe U of possible keys, storage can be reduced to $\Theta(|K|)$
-

Direct Addressing vs Hash Table

- Direct Address: an element with key k is stored in slot k.
    -
- Hash Table: an element with key k is stored in slot $h(k)$
- h maps the universe U of keys into the slots of a hash table $T[0..m-1]$

Summary

Helpful Notes: `http://www.cs.yale.edu/homes/aspnes/pinewiki/C(2f)HashTables.html?highlight=%28CategoryAlgorithmNotes%29`