

Hashing

Recap Problem: Flight Database

- One record of data:
 - Key
 - Data
- Dynamic collections
 - Can add or remove any one at anytime
- Can query the database
 - Find a particular record by the key
 - E.g. what is the flight at 09:05?
 - Or check the existence
 - Is there any flight between 09:00 to 09:20?
 - Find the one before or after
 - successor or predecessor

in a dictionary:
key = word
data = meaning

Time	To/From	Flight
08:20	LONDON	UL 125
08:45	NEW YORK	TH 9599
09:05	BARCELONA	AX 571
09:30	MOSCOW	BE 25836
09:55	DUBAI	LK 12121
10:20	PARIS	DM 7324
10:45	ROME	RS 1703
11:10	BERLIN	FX 50714

Dictionary ADT

<code>void insert(Key k, Value v)</code>	<i>insert (k,v) into table</i>
<code>Value search(Key k)</code>	<i>get value paired with k</i>
<code>Key successor(Key k)</code>	<i>find next key > k</i>
<code>Key predecessor(Key k)</code>	<i>find next key < k</i>
<code>void delete(Key k)</code>	<i>remove key k (and value)</i>
<code>boolean contains(Key k)</code>	<i>is there a value for k?</i>
<code>int size()</code>	<i>number of (k,v) pairs</i>

Examples

Dictionary: key = word
 value = definition

Phone Book key = name
 value = phone number

Internet DNS key = website URL
 value = IP address

C++ compiler key = variable name
 value = type and value

essentially just need to use a lookup table,
since these things do not have a successor to slowly
traverse down to find

Time Complexity for Each Operation

Data Structure

Linked List

Sorted Array

Unsorted Array

Balanced Tree

Query, Modification

$O(n)$

$O(\log n)$, $O(n)$

$O(n)$

$O(\log n)$

Can we do....?

$O(1)$



Dictionary/Symbol Tables

- Spelling correction (key=misspelled word, data=word)
- Scheme interpreter (key=variable, data=value)
- Web server
 - Lots of simultaneous network connections.
 - When a packet arrives, give it to the right process to handle the connection.
 - key=ip address, data = connection handler
- In these cases, $O(\log n)$ often isn't fast enough!

Assumptions

- No duplicate keys allowed.
- No mutable keys
 - If you use an object as a key, then you can't modify that object later.

```
SymbolTable<Time, Plane> t =  
    new SymbolTable<Time, Plane>();
```

```
Time t1 = new Time(9:00);  
Time t2 = new Time(9:15);
```

```
t.insert(t1, "SQ0001");  
t.insert(t2, "SQ0002");
```

```
t1.setTime(10:00);
```

```
x = new Time(9:00);  
t.search(x);
```

Moral: Keys should be immutable.

Examples: Integer, String

Attempt #1: Use a table, indexed by keys

- What I do after dinner in the seven days of a week

0	
1	
2	
3	
4	
5	
6	

Universe $U = \{0..6\}$ of size $m = 7$

(key, value)

(0, 'Netflix')

(3, 'Exercise')

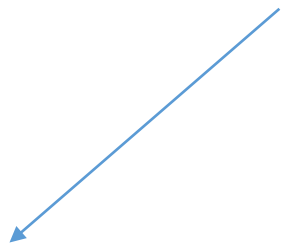
(5, 'Lecture')

Attempt #1: Use a table, indexed by keys

- What I do after dinner in the seven days of a week

0	'NetFlix'
1	
2	
3	'Exercise'
4	
5	'Lecture'
6	

Example: `insert(5, 'Lecture')`



Time:
insert: $O(1)$,search: $O(1)$

Direct Access Table

- Problems:
 - Too much space
 - If keys are ALL integers, then table-size > 4 billion
- What if keys are not integers?
 - Where do you put the key/value :
 “(hippopotamus, bob)”
 - Where do you put 3.14159?

Pythagoras said, “Everything is a number.”



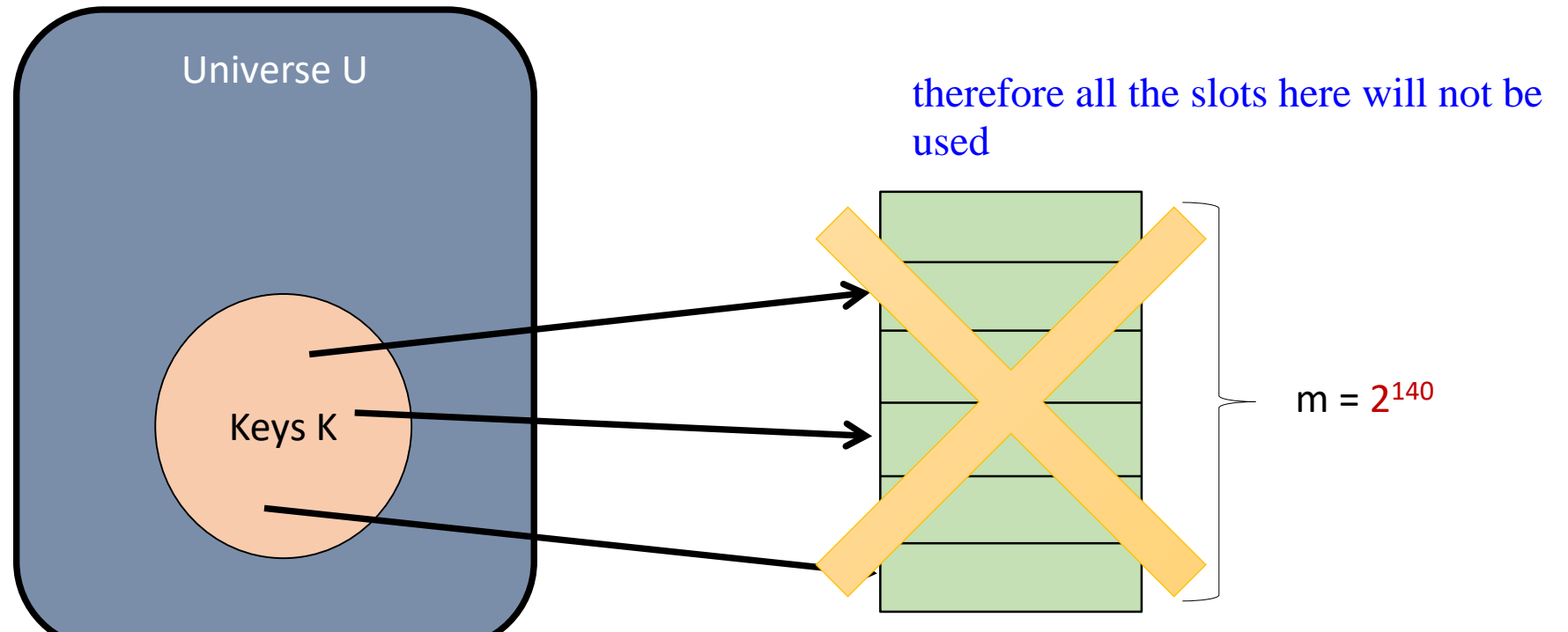
Direct Access Tables

- Pythagoras said, “Everything is a number.”
 - Everything is just a sequence of bits.
 - Treat those bits as a number.
- English:
 - 26 letters => 5 bits/letter
 - Longest word = 28 letters (antidisestablishmentarianism?)
 - 28 letters * 5 bits = 140 bits
 - So we can store any English text in a direct-access array of size 2^{140} .
 - \approx number of atoms in observable universe

Hash Functions

- Problem:

- Huge universe U of possible keys.
- Smaller number n of actual keys.
- We cannot have an array with size $m = 2^{140}$



Pythagoras said, “Everything is a number.”

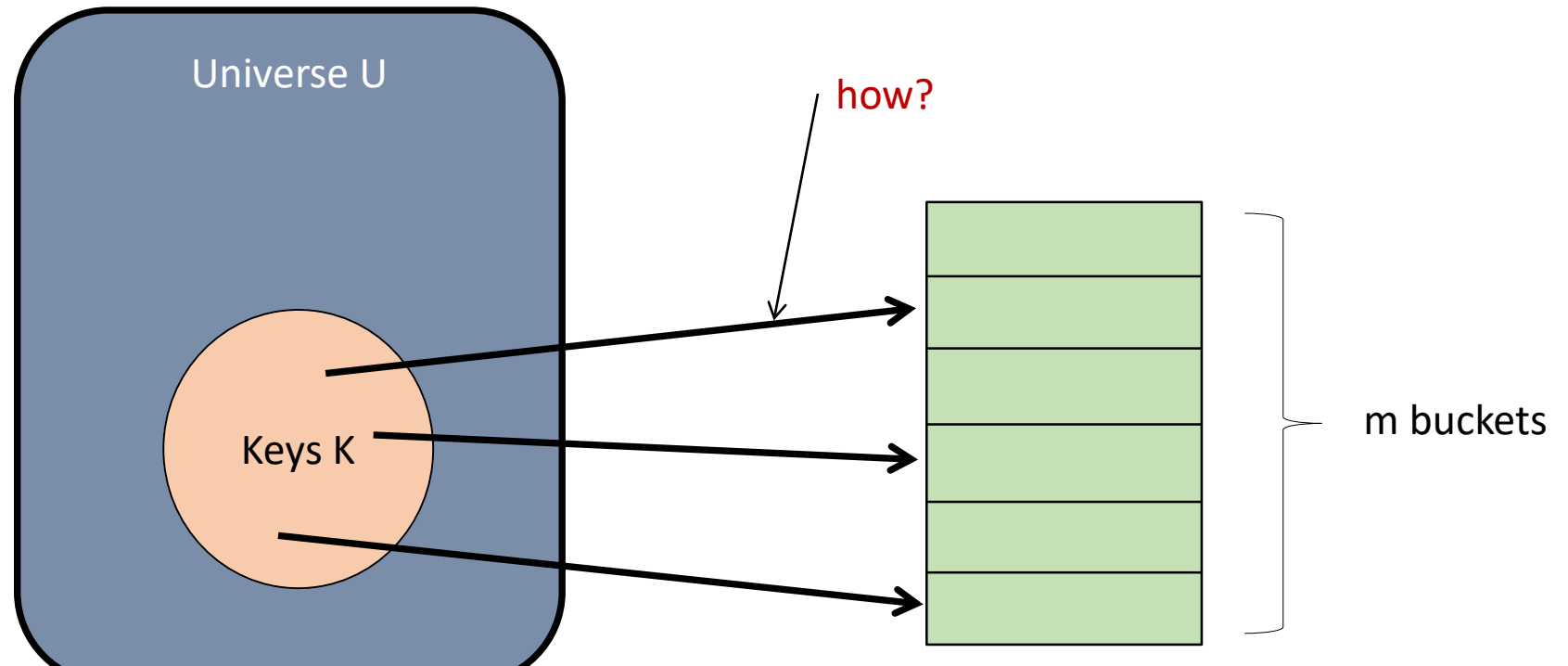
- English words:
 - There are possible 2^{140} English words with less than or equal to 28 letters
 - But we actually have about 1 million ($\sim 2^{20}$) of REAL English words
 - Comparing to 2^{140}

Hash Functions

therefore hash functions is finding another way to represent the current value

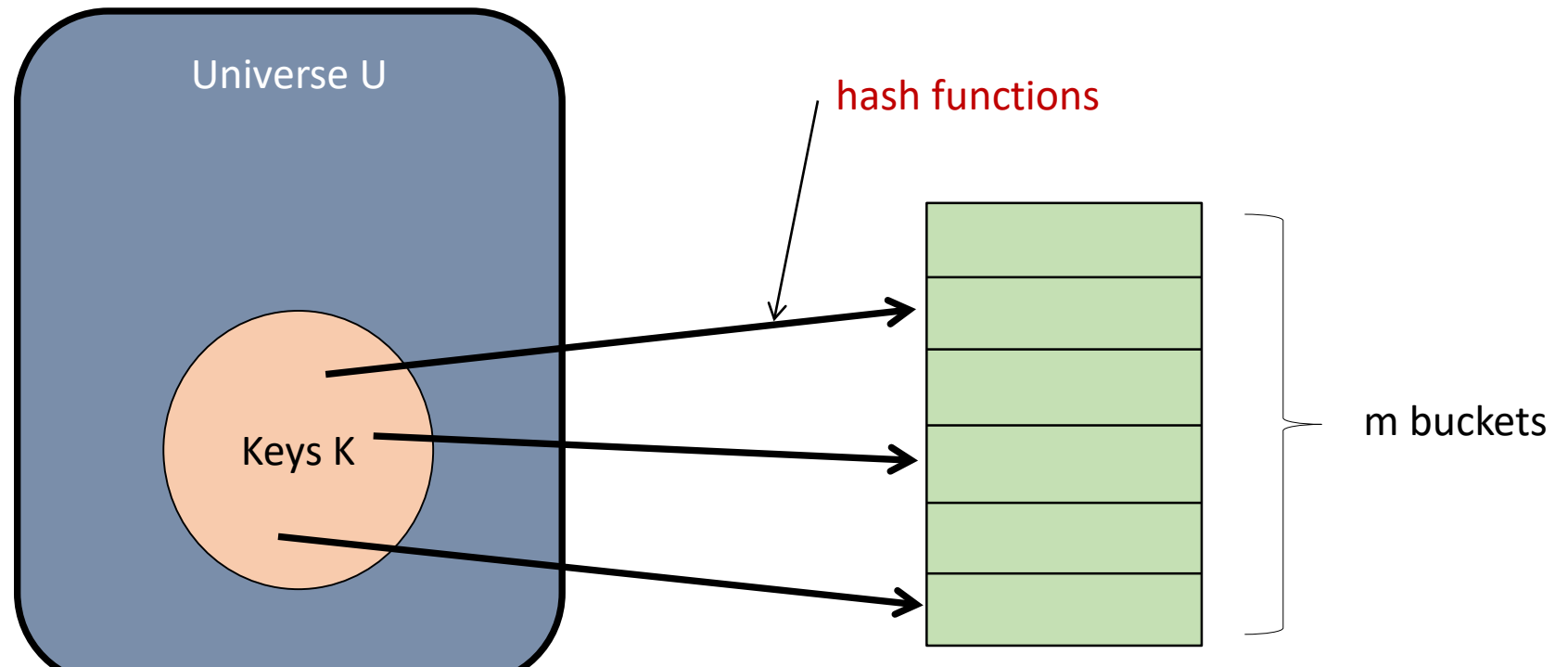
- Problem:

- Huge universe U of possible keys. ← e.g., $u = 2^{140}$
- Smaller number n of actual keys. ← e.g., $n = 2^{20}$
- How to put n items into, say $m \approx n$ buckets?



Hash Functions

- Define hash function $h : U \rightarrow \{1..m\}$
 - Store key k in bucket $h(k)$.
 - Time complexity:
 - Time to compute h + Time to access bucket
 - For now: assume hash function cost $O(1)$ to compute.



Hash Functions

- For example, the (key, value) pairs are:
 - ("pizza", "Clementi")
 - ("coffee", "NUS")
- For example, if the function h is the number of characters in an English word then
 - $h(\text{"pizza"}) = 5$
 - $h(\text{"coffee"}) = 6$
- What is the potential problem?

0	null
1	null
2	null
3	null
4	null
5	("pizza","Clementi")
6	("orange","NUS")
7	null
8	null
9	null

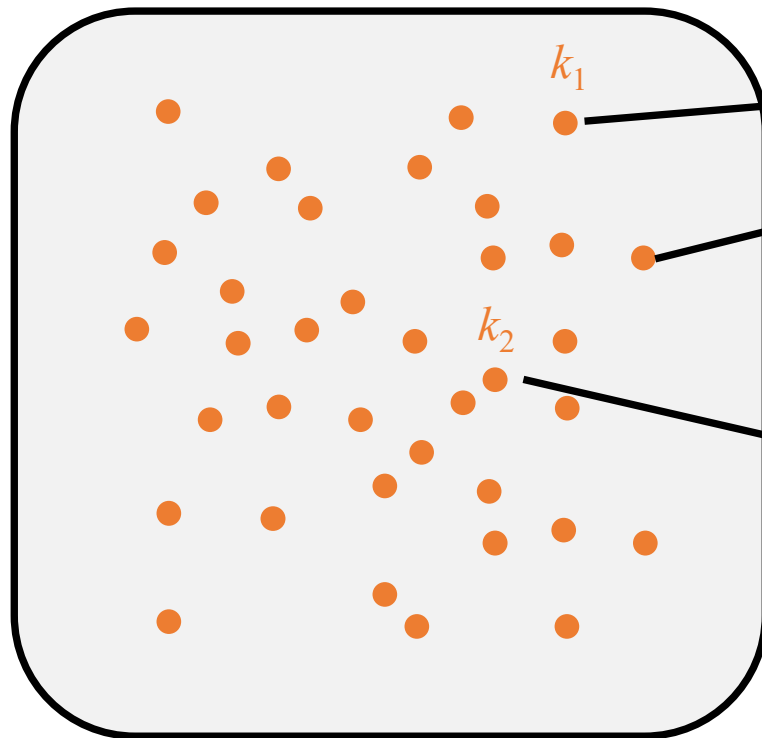
Hash Functions

$\text{insert}(k_1, A)$

$\text{insert}(k_2, B)$

$\text{insert}(k_3, C)$

Collision!



$h(k_1) = 2$

$h(k_3) = 2$

$h(k_2) = 8$



0

1

2

3

4

5

6

7

8

9

null

A

null

null

null

null

null

B

null

Hash Collisions

- We say that two distinct keys k_1 and k_2 collide if:

$$h(k_1) = h(k_2)$$

- For example, if the function h is the number of characters in an English word then

$$h(\text{"pizza"}) = h(\text{"mango"}) = 5$$

Can we choose a hash function with no collisions?

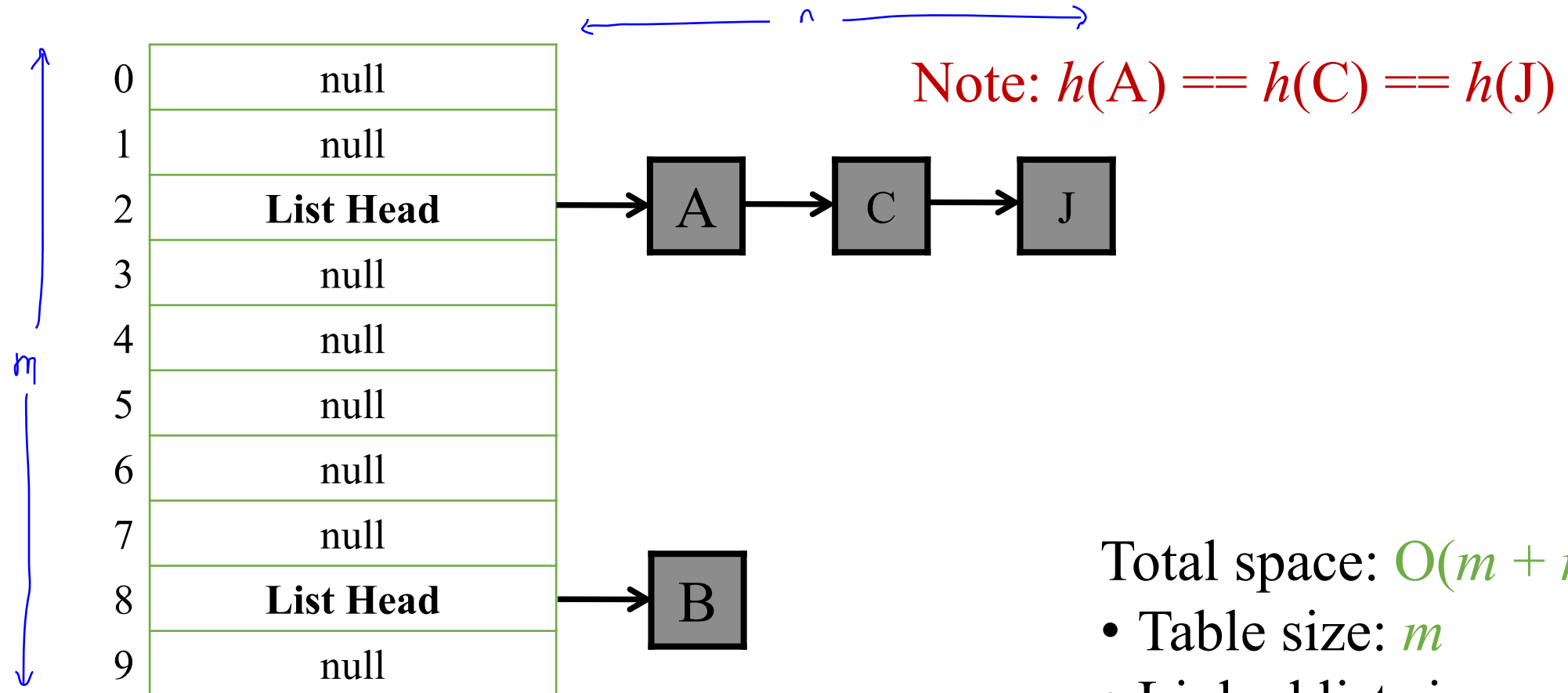
- Unavoidable!
 - The table size is smaller than the universe size.
- The pigeonhole principle says:
 - There must exist two keys that map to the same bucket.
 - Some keys must collide!

Coping with Collision

- Idea 1: choose a new, better hash functions
 - Hard to find.
 - Requires re-copying the table.
 - Eventually, there will be another collision.
- Idea 2: chaining
 - Put both items in the same bucket! similar to linked list
- Idea 3: open addressing
 - Find another bucket for the new item.

Chaining

- Each bucket contains a linked list of items.



Hashing with Chaining

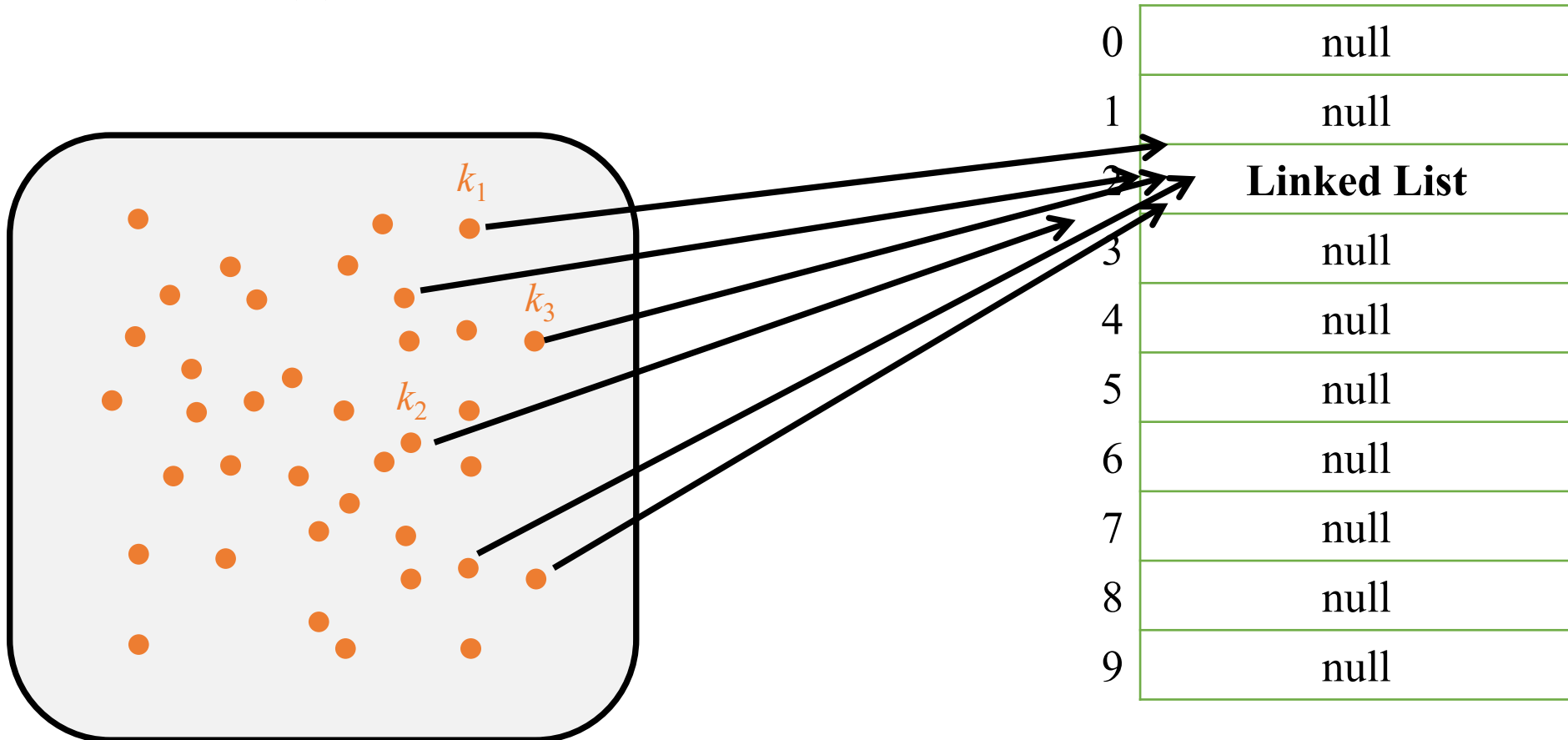
- Insertion $O(1)$
 - `insert(key, value)`
 - Calculate `h(key)` $O(1)$
 - Lookup `h(key)` and add (key, value) to the linked list. $O(1)$
- search(key)/deletion
 - Calculate `h(key)`
 - Search for (key, value) in the linked list.
- Worst case?
 - time depends on length of linked list

Reminds me of this



Hashing with Chaining

- Assume all keys hash to the same bucket!
 - Search costs $O(n)$

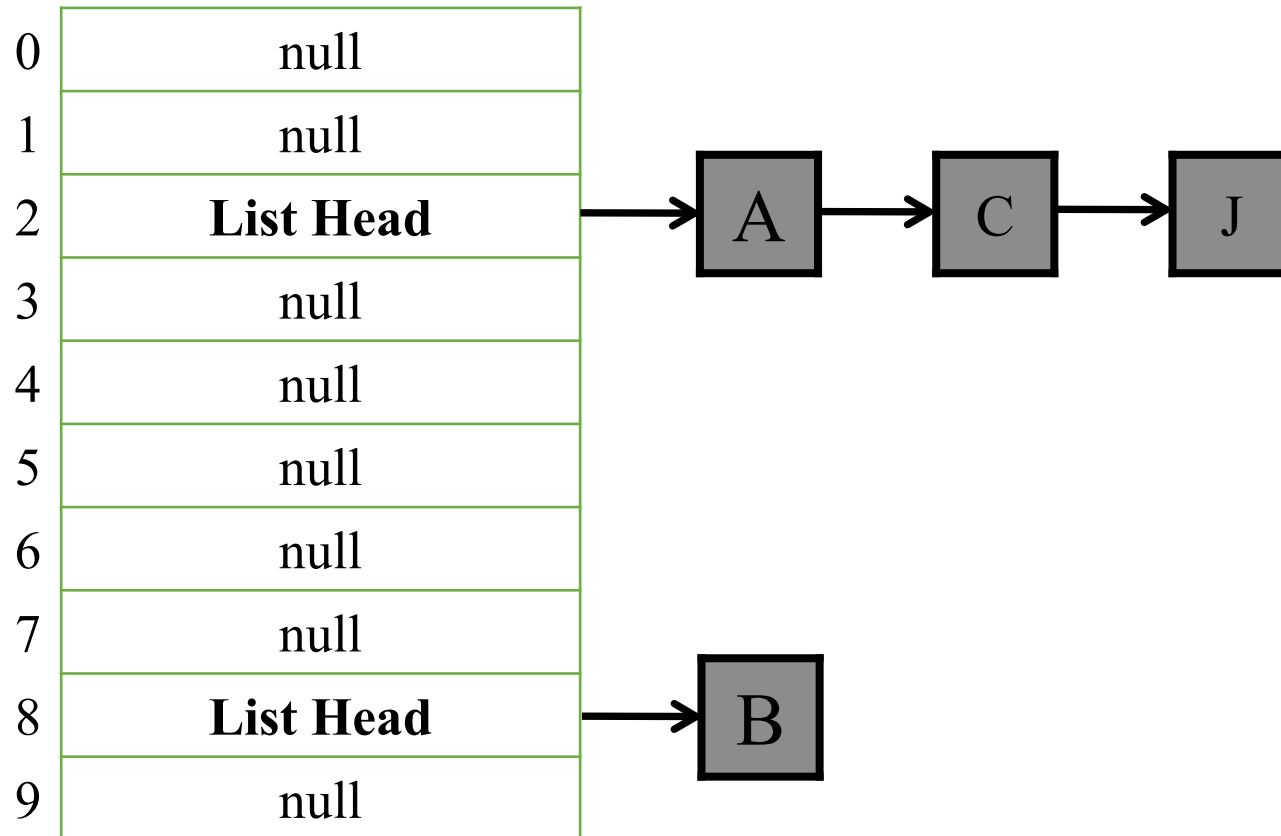


Let's be optimistic first....

- The Simple Uniform Hashing Assumption
 - Every key is equally likely to map to every bucket.
 - Keys are mapped independently.
- Intuition:
 - Each key is put in a random bucket.
 - Then, as long as there are enough buckets, we won't get too many keys in any one bucket.

A little probability

- What is the expected number of items in a bucket?



Let's be optimistic today.

- The Simple Uniform Hashing Assumption

- Assume:

- n items
 - m buckets

- Define: $\text{load}(\text{hash table}) = n/m$
= average # items / buckets.

- Expected search time = $1 + n/m$

hash function + array access

linked list traversal

- If $m > n$

- Expected search time = $O(1)$



Hashing with Chaining

- Searching:

- Expected search time = $1 + n/m = O(1)$
- Worst-case search time = $O(n)$

- Inserting:

- Worst-case insertion time = $O(1)$

Reality Fights Back

- Simple Uniform Hashing doesn't exist.
- Keys are not random.
 - Lots of regularity.
 - Mysterious patterns.
- Patterns in keys can induce patterns in hash functions unless you are very careful.

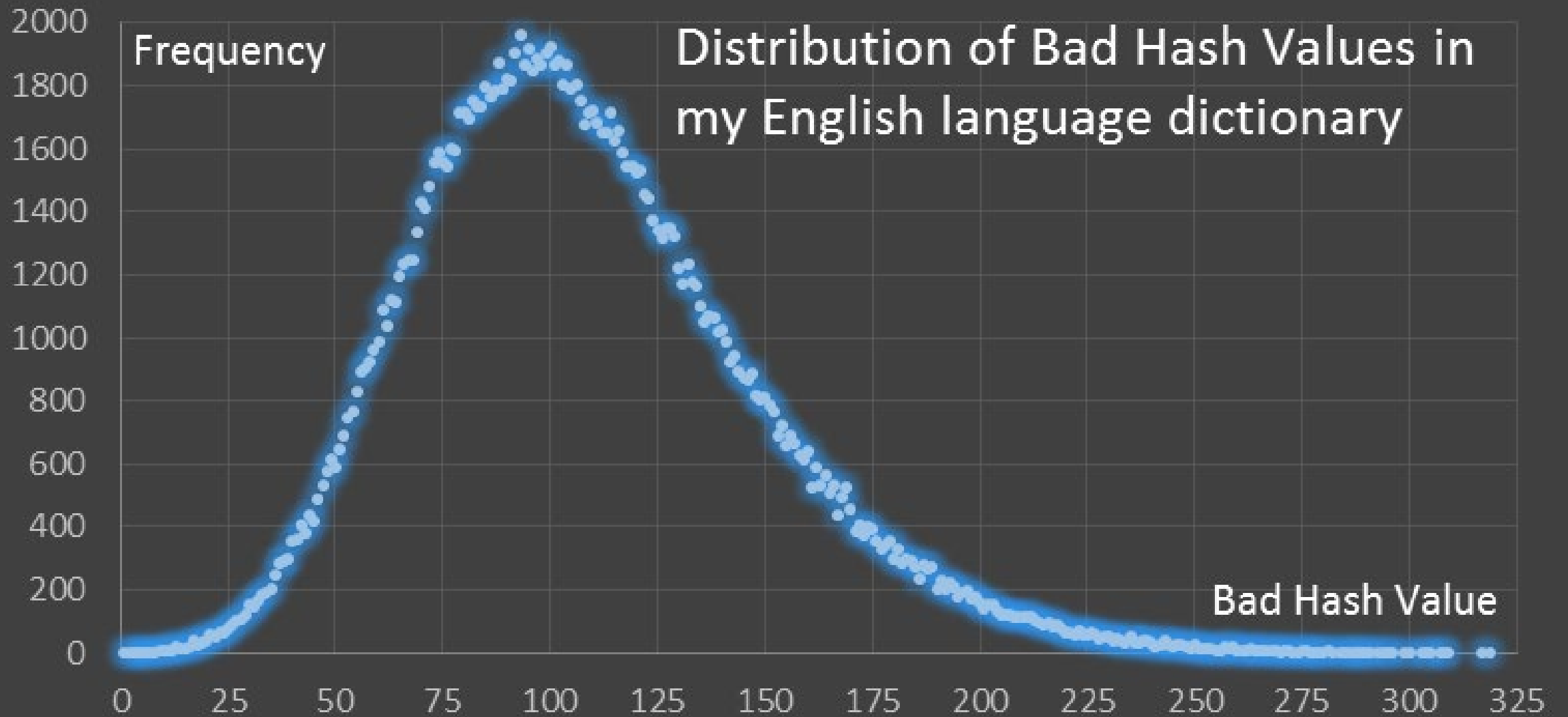
Example

- One bucket for each letter [a..z]
- Hash function: `h (string) = first letter`.
 - E.g., `h ("hippopotamus") = h`.
- Bad hash function: many fewer words start with the letter x than start with the letter s.

Example

- One bucket for each number from $[1..26*28]$
- Hash function: $h(\text{string}) = \text{sum of the letters}$.
 - E.g., $h(\text{"hat"}) = 8 + 1 + 20 = 29$.
- Bad hash function: lots of words collide, and you don't get a uniform distribution (since most words are short).

Distribution of Bad Hash Values in
my English language dictionary



Moral of the Story

- Don't design your own hash functions.
 - Ever.
- Unless you really need to.
- But pretty good hash functions do exist...
 - Optimism pays off!

Designing Hash Functions (If you really have to)

- Two common hashing techniques...
 - Division Method
 - Multiplication Method

Division Method

- $h(k) = k \bmod m$
 - For example: if $m = 7$, then $h(17) = 3$
 - For example: if $m = 20$, then $h(100) = 0$
 - For example: if $m = 20$, then $h(97) = 17$
- Two keys k_1 and k_2 collide when:
$$k_1 \bmod m = k_2 \bmod m$$
- Collision unlikely if keys are random.

Division Method Problem: Regularity

- What if k and m has a common divisor d ?

$$k = k \bmod m + i * m$$

↑ ↑
divisible by d divisible by d

- Implies that $h(k) = k \bmod m$ is divisible by d .
- For all those key values that are divisible by d , by what fraction of the hash table will they utilize?

$$1/d$$

0	A
1	null
2	null
$d=3$	B
4	null
5	null
$2d=6$	C
7	null
8	null
$3d=9$	D

Division Method

division method problem is finding a good prime number m

- $h(k) = k \bmod m$
- Choose $m =$ prime number
- Division method is popular (and easy), but not always the most effective.
 - Division is slow.

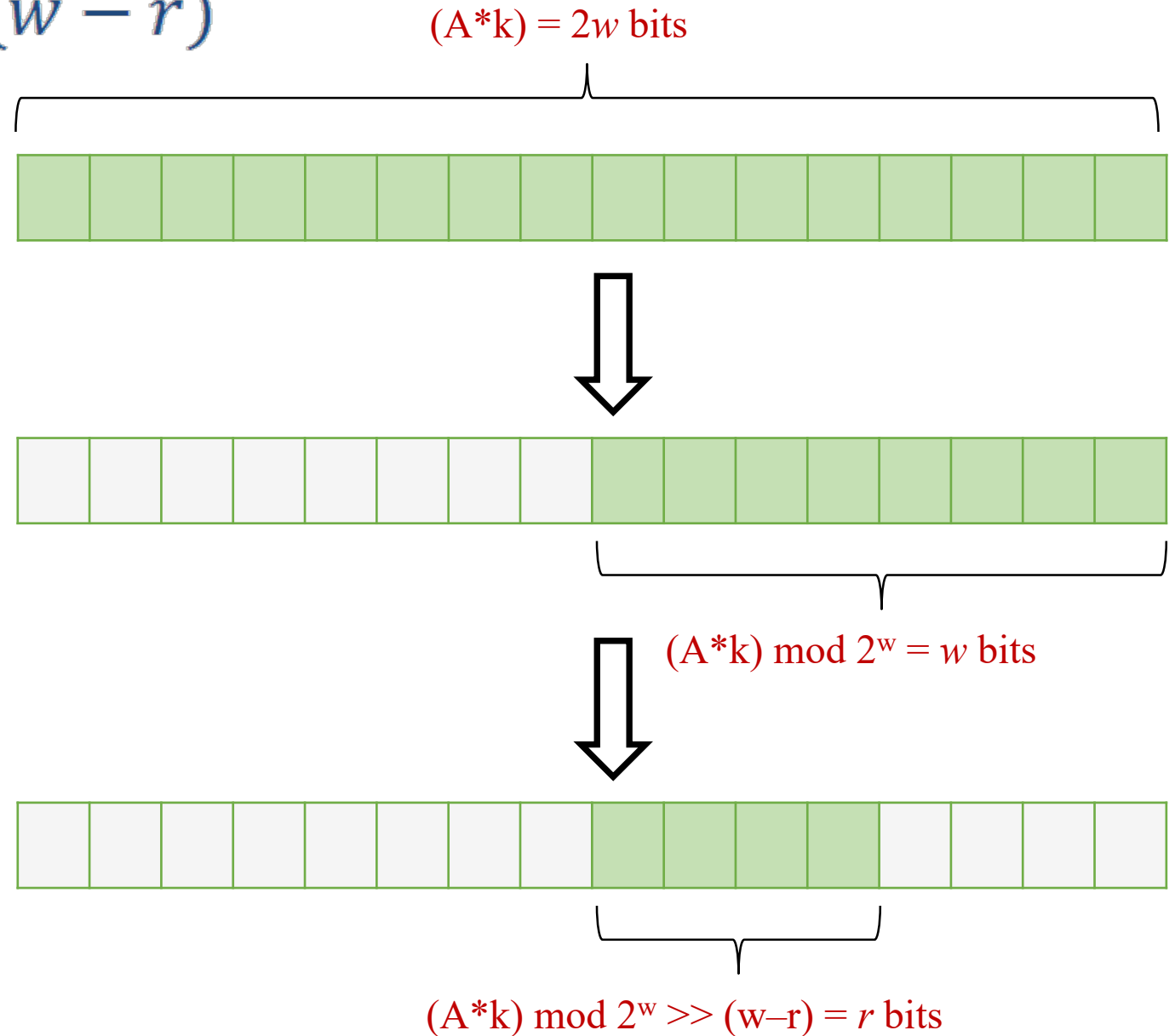
Multiplication Method

- Fix table size: $m = 2^r$, for some constant r .
- Fix word size: w , size of a key in bits.
- Fix (odd) constant A .

$$h(k) = (Ak) \bmod 2^w \gg (w - r)$$

$$h(k) = (Ak) \bmod 2^w \gg (w - r)$$

- A and k are w bits integers



Multiplication Method

problem is A

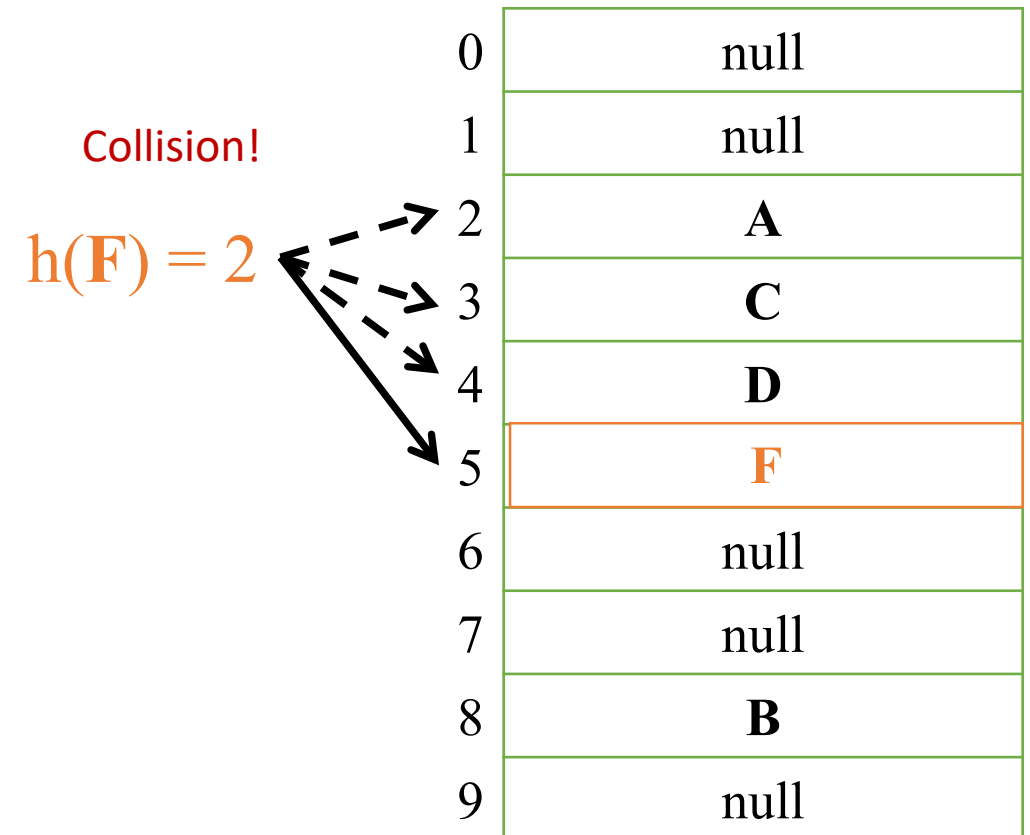
- Faster than Division Method
 - Multiplication, shifting faster than division
- Works reasonably well when A is an odd integer $> 2^{w-1}$
 - Odd: if it is even, then lose at least one bit's worth
 - Big enough: use all the bits in A .

Coping with Collision

- Idea 1: choose a new, better hash functions
 - Hard to find.
 - Requires re-copying the table.
 - Eventually, there will be another collision.
- Idea 2: chaining
 - Put both items in the same bucket!
- Idea 3: open addressing
 - Find another bucket for the new item.

Open Addressing

- Advantages:
 - No linked lists!
 - All data directly stored in the table.
 - One item per slot.
- On collision
 - **Probe** a sequence of buckets until you find an empty one.



Probing

- Find the next position that's empty to insert a key
- If the current slot with index i in the table is occupied, try slot $i + 1$
- It is the same way to say
 - originally let's try $h(F)$
 - if it collides, try $(h(F) + 1) \bmod m$ ← After 1 collision
 - if it collides again, try $(h(F) + 2) \bmod m$ ← After 2 collision
 - if it collides again, try $(h(F) + 3) \bmod m$ ← After 3 collision
 - ...

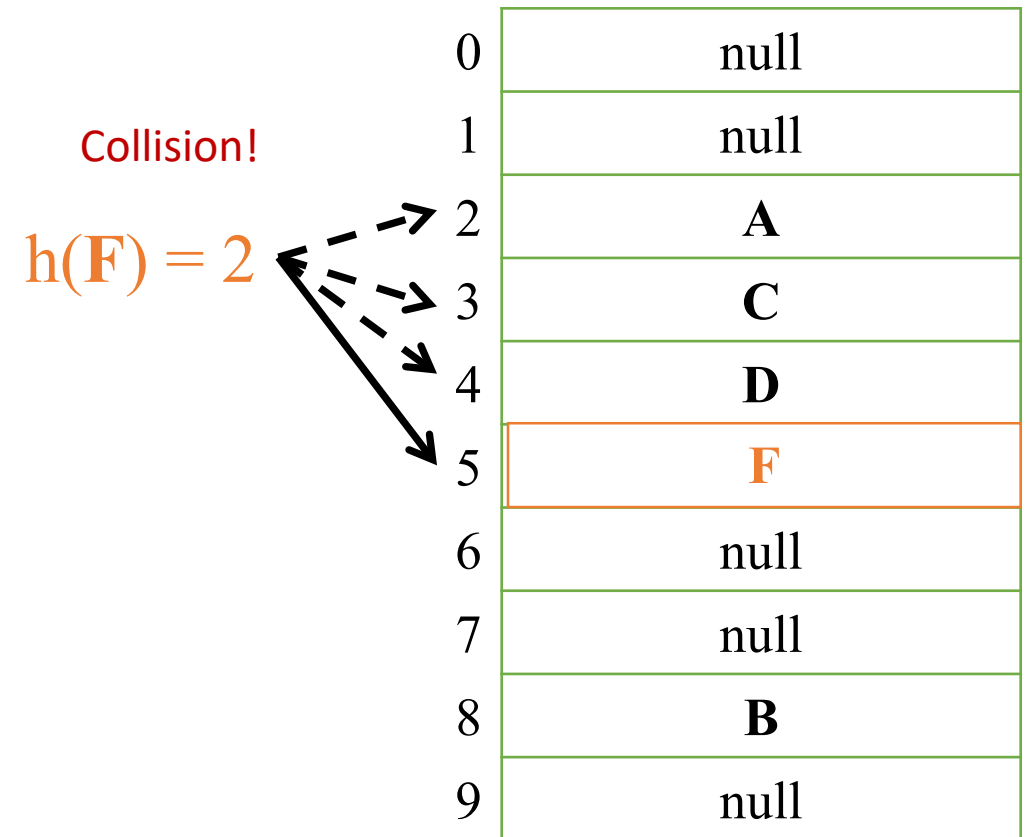
Collision!

$h(F) = 2$

0	null
1	null
2	A
3	C
4	D
5	F
6	null
7	null
8	B
9	null

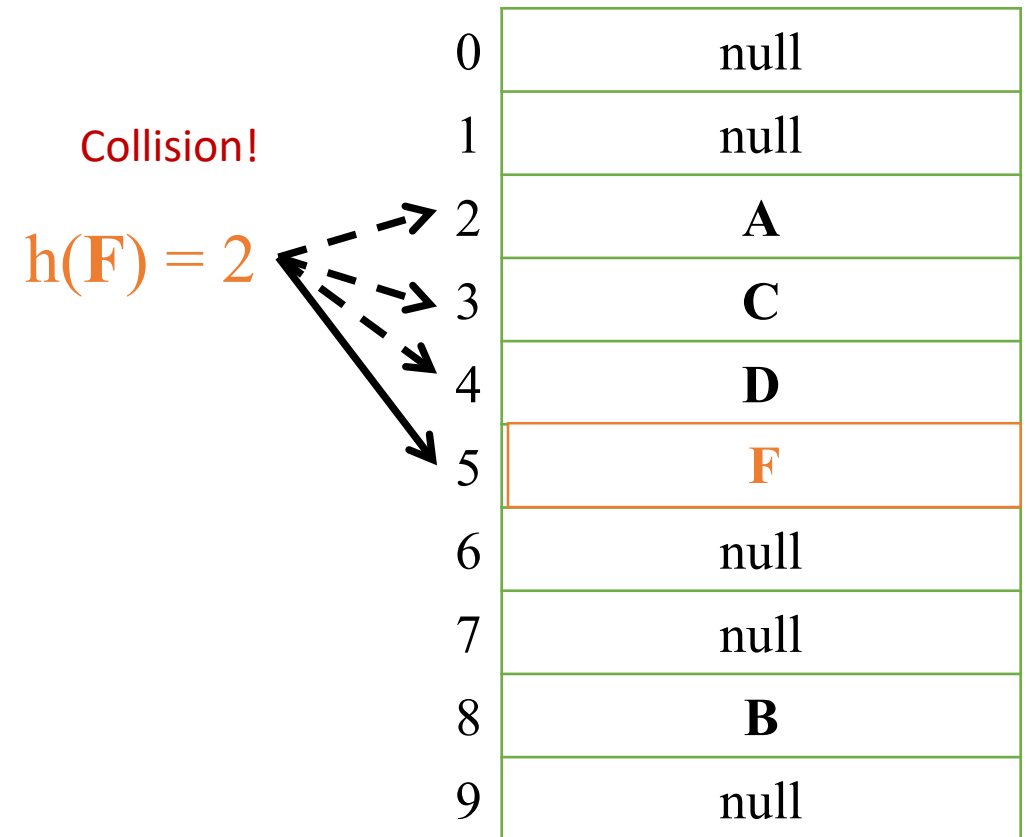
Probing

- Find the next position that's empty to insert a key
- If the current slot with index i in the table is occupied, try slot $i + 1$
- It is the same way to say
 - originally let's try $h(F)$
 - if it collides, try $(h(F) + i) \bmod m$
 - After i collisions
- Or if it collides, try $(h(F) + f(i)) \bmod m$
 - After i collisions
 - for $f(i) = i$



Probing

- Find the next position that's empty to insert a key
- originally let's try $h(F)$
- if it collides, try $(h(F) + f(i)) \bmod m$
 - After i collisions
 - for $f(i) = i$
- We have freedom to change $f(i)$ into other functions
- For $f(i) = i$, it is called **Linear Probing**



Probing

- For $f(i) = i^2$, it is called **Quadratic Probing**
- Originally let's try $h(F)$
- if it collides, try $(h(F) + 1^2) \bmod m$
- if it collides again, try $(h(F) + 2^2) \bmod m$
- if it collides again, try $(h(F) + 3^2) \bmod m$
- ...
- After i collisions, try $(h(F) + i^2) \bmod m$

Collision!
 $h(F) = 2$

0	null
1	null
2	A
3	C
4	D
5	null
6	F
7	null
8	B
9	null

Some Probing Functions

- Let's redefine the hashing function to be $h(\text{key}, i)$ for i is the number of collisions
- Linear probing:

$$h(\text{key}, i) = h(\text{key}) + i$$

- Quadratic probing

$$h(\text{key}, i) = h(\text{key}) + i^2$$

- Double hashing (for another hashing function g)

$$h(\text{key}, i) = h(\text{key}) + i \times g(\text{key})$$

How do we search now?

pseudo code for searching

```
int i = 0;
while (i <= m) {
    int bucket = h(key, i);
    if (T[bucket] == null) // Empty bucket!
        return key-not-found;
    if (T[bucket].key == key) // Full bucket.
        return T[bucket].data;

    i++;
}
return key-not-found; // Exhausted entire table.
```

$h(F) = 2$



0	null
1	null
2	A
3	C
4	D
5	F
6	null
7	null
8	B
9	null

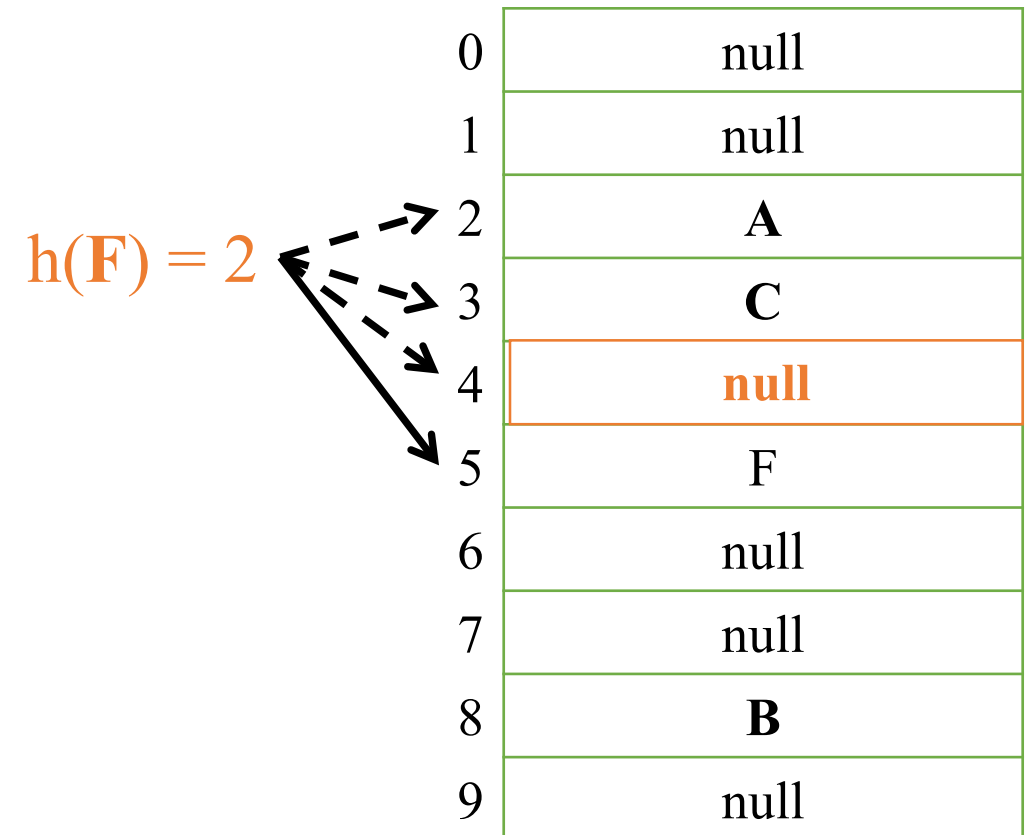
worst case is $O(n)$?

Open Addressing with Probing

- Now we can
 - Insert
 - Search
- How about deleting an item?
 - Just set the slot to “null”?
 - E.g. `delete(D)` ?
 - Problem?

if set to null after deletion, then it will be seen as the bucket has not been used before

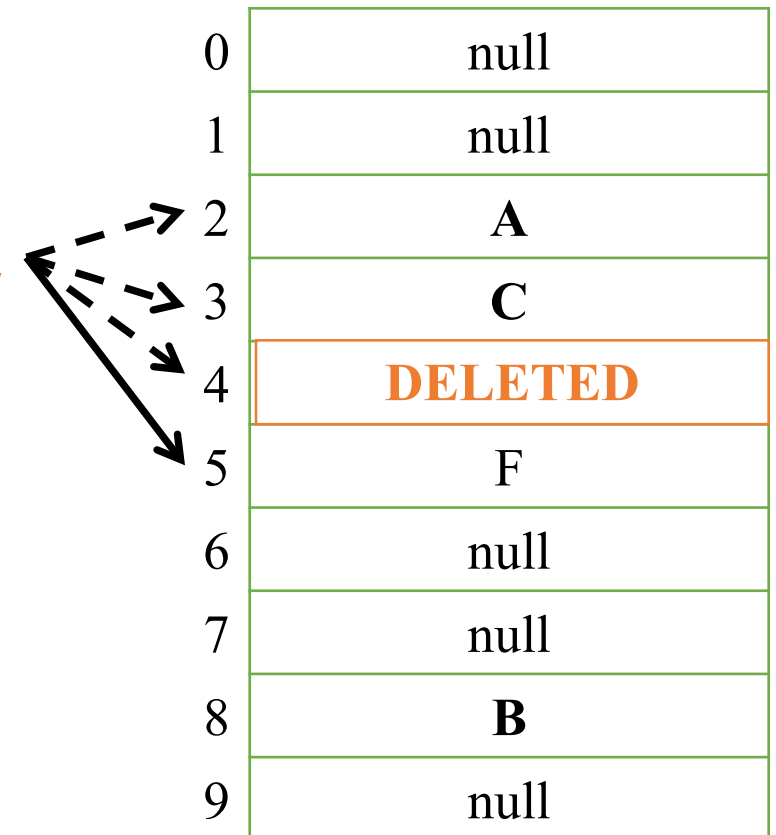
-will thus not continue to probe



Open Addressing with Probing

- Deletion:
 - Set the slot to be “DELETED”
- For Searching, we can use the previous algorithm
 - Note that we will stop searching when we met a “null” or we found the key ONLY
- How about insertion now?
 - We can replace an item on the slot of “DELETED”

$$h(F) = 2$$

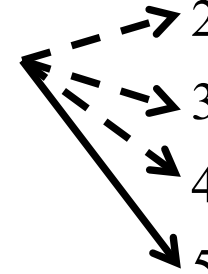


0	null
1	null
2	A
3	C
4	DELETED
5	F
6	null
7	null
8	B
9	null

Deletion Example

- Assuming $h(F,0) = 2$
- Delete(D)
- Search for F
 - Same

$h(F) = 2$

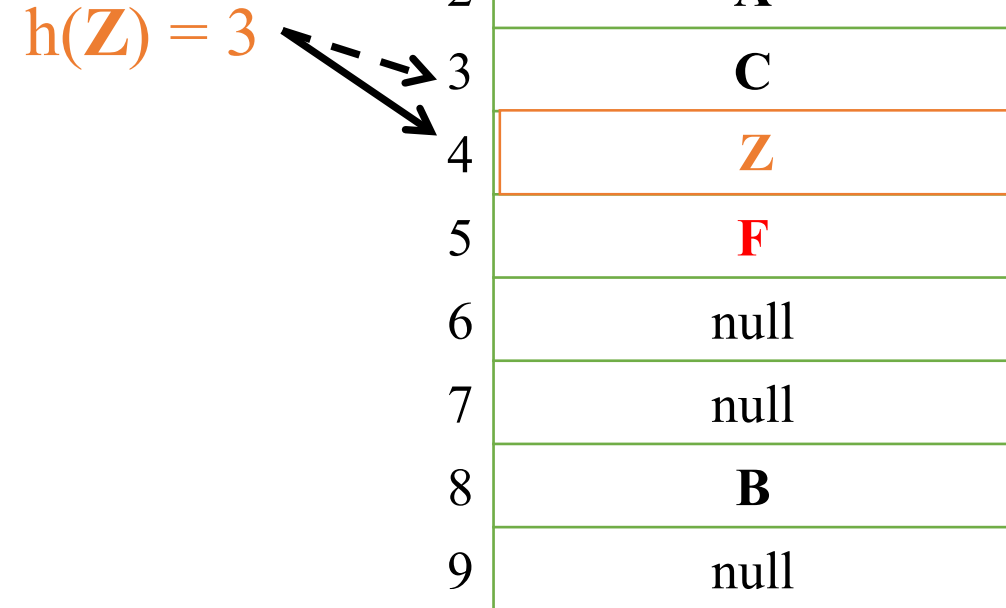


0	null
1	null
2	A
3	C
4	DELETED
5	F
6	null
7	null
8	B
9	null

Deletion Example

- Assuming $h(F,0) = 2$
- Delete(D)
- Search for F
 - Same
- Now insert Z for $h(Z) = 3$
 - Collided with C
 - Look for $h(Z,1) = 4$
 - Insert Z into slot 4

$$h(Z) = 3$$



0	null
1	null
2	A
3	C
4	Z
5	F
6	null
7	null
8	B
9	null

Two Properties of Good Hashing Functions

1. $h(\text{key}, i)$ must be able to reach all slots

- For every bucket j , there is some i such that:

$$h(\text{key}, i) = j$$

- For linear probing: true!

2. Simple Uniform Hashing Assumption

- Every key is equally likely to be mapped to every bucket, independently of every other key.
- An “Art”
- Could be either with good math or empirical proof

Performance

Performance

- Under uniform hashing assumption, what is the expected time for insertion after we inserted m items into a table of size m ?

chaining

a) $O(1)$

b) $O(\log n)$

c) $O(n)$

d) $O(n^2)$

e) None of the above.

different answers for chaining and open addressing

open addressing - since it is full

Performance of Open Addressing

- Chaining:
 - When $m == n$, we can still add new items to the hash table.
 - We can still search efficiently.
- Open addressing:
 - When $m == n$, the table is full. assumption is that $m > n$
 - We cannot insert any more items.
 - We cannot search efficiently.

Performance of Open Addressing

- Define:
 - Load: $\alpha = n / m$
 - Assume $\alpha < 1$.
- Claim:
- For n items, in a table of size m , assuming uniform hashing, the expected cost of an operation is:

$$\leq \frac{1}{1 - \alpha}$$

- Example: if ($\alpha = 90\%$), then $E[\# \text{ probes}] = 10$

Performance of Open Addressing

- First probe: probability that first bucket is full is:

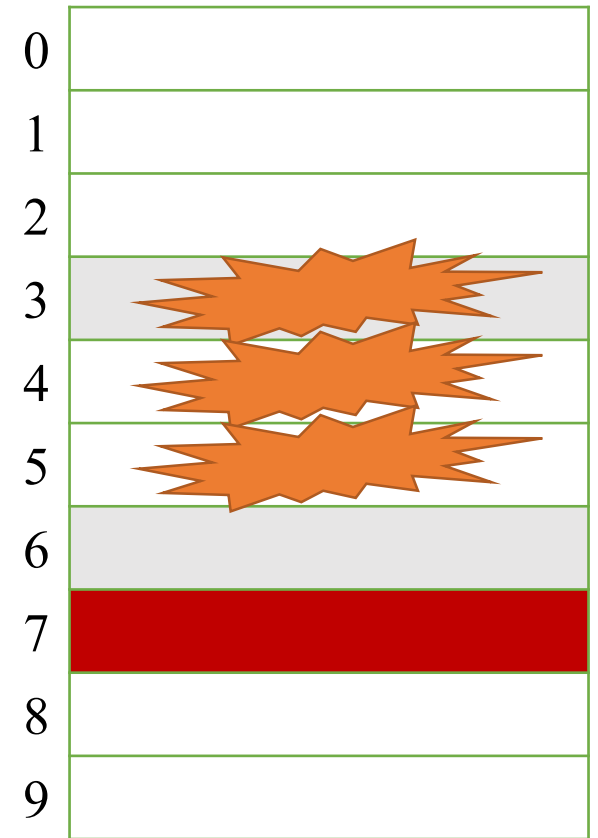
$$n / m$$

- Second probe: if first bucket is full, then the probability that the second bucket is also full:

$$(n - 1) / (m - 1)$$

- Third probe: probability is full:

$$(n - 2) / (m - 2)$$



Performance of Open Addressing

- Expected #probes

$$1 + \frac{n}{m} \left(\text{Expected cost of remaining probes} \right)$$

First probe

Probability of collision on first probe

Performance of Open Addressing

- Expected #probes

$$1 + \frac{n}{m} \left(\text{Expected cost of remaining probes} \right)$$

$$1 + \frac{n}{m} \left(1 + \frac{n-1}{m-1} \left(\text{Expected cost of remaining probes} \right) \right)$$

First probe

Probability
of collision
on first probe

Probability of collision
on second probe

Performance of Open Addressing

- Expected #probes

$$1 + \frac{n}{m} \left(1 + \frac{n-1}{m-1} \left(1 + \frac{n-2}{m-2} \left(1 + \frac{n-3}{m-3} (\dots) \right) \right) \right)$$

- Note that

$$\frac{n-i}{m-i} \leq \frac{n}{m} \leq \alpha$$

Performance of Open Addressing

- Expected #probes

$$1 + \frac{n}{m} \left(1 + \frac{n-1}{m-1} \left(1 + \frac{n-2}{m-2} \left(1 + \frac{n-3}{m-3} (\dots) \right) \right) \right)$$

$$\leq 1 + \alpha \left(1 + \alpha \left(1 + \alpha \left(1 + \alpha (\dots) \right) \right) \right)$$

$$\leq 1 + \alpha + \alpha^2 + \alpha^3 + \alpha^4 + \dots$$

$$\leq \frac{1}{1 - \alpha}$$

Performance of Open Addressing

- Define:
 - Load: $\alpha = n / m$
 - Assume $\alpha < 1$.
- Claim:
- For n items, in a table of size m , assuming uniform hashing, the expected cost of an operation is:

$$\leq \frac{1}{1 - \alpha}$$

- Example: if ($\alpha = 90\%$), then $E[\# \text{ probes}] = 10$

Open Addressing Advantages

- Saves space
 - Empty slots vs. linked lists.
- Rarely allocate memory
 - No new list-node allocations.
- Better cache performance
 - Table all in one place in memory
 - Fewer accesses to bring table into cache.
 - Linked lists can wander all over the memory.

Open Addressing Disadvantages

- More sensitive to choice of hash functions.
 - Clustering is a common problem.
 - See issues with linear probing.
- More sensitive to load.
 - Performance degrades badly as $\alpha \rightarrow 1$.

