CS2040S
Data Structures and Algorithms

Hashing!
(Part 1)

Announcement

- No lecture on 3 March (Monday of Week 7)

- Instead, Midterm on 3 March scheduled between 6:30 pm – 9 pm at MPSH 2A/B

  1 Page 2-sided A4 cheat sheet

  Topics: Up until last lecture

Sample paper will be uploaded on Coursemology

Announcement

PSet 5 Release!

Implementing other forms of trees!

- AI related and autocomplete applications based on the trees.

Due 23:59 Sunday of Week 7. (Half a week of extra time)

Plan: today and next

Three (or Four) Days of Hashing

- Applications
- Basic theory
- Handling collisions
- (Hashing in Java)
- Amortized analysis (doubling/shrinking)
- Sets and Bloom filters

Topic of today: Hash Tables

Abstract Data Types

Symbol Table

public interface SymbolTable

`void`	`insert(Key k, Value v)`	*insert (k,v) into table*
`Value`	`search(Key k)`	*get value paired with k*
`void`	`delete(Key k)`	*remove key k (and value)*
`boolean`	`contains(Key k)`	*is there a value for k?*
`int`	`size()`	*number of (k,v) pairs*

Note: no successor / predecessor queries.

Symbol Table

Examples:

Dictionary: key = word

value = definition

Phone Book key = name

value = phone number

Internet DNS key = website URL

value = IP address

Java compiler key = variable name

value = type and value

Implement symbol table with an AVL tree:
(C_I= cost insert, C_S=cost search)

1. $C_I = O(1)$, $C_S = O(1)$
2. $C_I = O(1)$, $C_S = O(\log n)$
3. $C_I = O(1)$, $C_S = O(n)$
4. ✓ $C_I = O(\log n)$, $C_S = O(\log n)$
5. $C_I = O(n)$, $C_S = O(\log n)$
6. $C_I = O(n)$, $C_S = O(n)$

Symbol Table

Implement a symbol table with:

- $C_I = O(1)$
- $C_S = O(1)$

Fast, fast, fast….

Dictionaries vs. Symbol Tables

What can you do with a dictionary but not a symbol table?

Dictionaries vs. Symbol Tables

Sorting with a dictionary:

1) Insert every item into the dictionary.
2) Search for the minimum item.
3) Repeat: find successor

Running time to implement sorting:

With an AVL tree/dictionary?

Dictionaries vs. Symbol Tables

Sorting with a dictionary:

1) Insert every item into the dictionary.

2) Search for the minimum item.

3) Repeat: find successor

Running time to implement sorting:

With an AVL tree/dictionary? O(n log n)

With a symbol table?

Dictionaries vs. Symbol Tables

Sorting with a dictionary:

1) Insert every item into the dictionary.

2) Search for the minimum item.

3) Repeat: find successor

Running time to implement sorting:

With an AVL tree/dictionary? O(n log n)

With a symbol table? $O(n^2)$

- No efficient way to find minimum item!
- No ordering of elements.

Sorting (aside)

Isn't O(1) search/insert impossible?

- Sorting takes $\Omega(n \log n)$ comparisons.

Sorting (aside)

Isn't O(1) search/insert impossible?

Sorting takes $\Omega(n \log n)$ comparisons.

- How do you sort with a symbol table?
- Only search/insert/delete.

Sorting (aside)

Isn't O(1) search/insert impossible?

Sorting takes $\Omega(n \log n)$ comparisons.

- How do you sort with a symbol table?
- Only search/insert/delete.

(Binary) search takes $\Omega(\log n)$ comparisons.

- Impossible to search in fewer than $\log(n)$ comparisons.
- But a symbol table finds an item in O(1) steps!!
- Conclusion: symbol table is not ***comparison-based***.

Building a Symbol Table

Direct Access Tables

Attempt #1: Use a table, indexed by keys.

0	null
1	null
2	item1
3	null
4	null
5	item3
6	null
7	null
8	item2
9	null

Universe U={0..9} of size m = 10.

<u>(key, value)</u>

(2, item1)
(8, item2)
(5, item3)

Assume keys are distinct.

Direct Access Tables

Attempt #1: Use a table, indexed by keys.

0	null
1	null
2	item1
3	null
4	null
5	item3
6	null
7	null
8	item2
9	null

Example: insert(4, Seth)

Direct Access Tables

Attempt #1: Use a table, indexed by keys.

0	null
1	null
2	item1
3	null
4	Seth
5	item3
6	null
7	null
8	item2
9	null

Example: insert(4, Seth)

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

Direct Access Tables

Problems:

- What if keys are not integers?
  - Where do you put the key/value "**(hippopotamus, bob)**"?
  - Where do you put 3.14159…?

Direct Access Tables

Pythagoras said, “Everything is a number.”

“The School of Athens” by Raphael

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 word in a direct-access array of size 2^{140}.

Direct Access Tables

Pythagoras said, "Everything is a number."

– Everything is just a sequence of bits.

– Treat those bits as a number.

have a 140-bit long array. The i^{th} position is for the i^{th} word.

If the i^{th} bit is 1, the word is stored. Otherwise it is not stored.

– English:

- 26 letters => 5 bits/letter
- Longest word = 28 letters (antidisestablishmentarianism?)
- 28 letters * 5 bits = 140 bits
- So we can store any English word in a direct-access array of size 2^{140}.

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 word in a direct-access array of size 2^{140}. ≈ number of atoms in observable universe

Direct Access Tables

Problems:

- What if keys are not integers?
  - Where do you put the key/value “**(hippopotamus, bob)**”?
  - Where do you put 3.14159…?
  - Can represent anything as a sequence of bits.

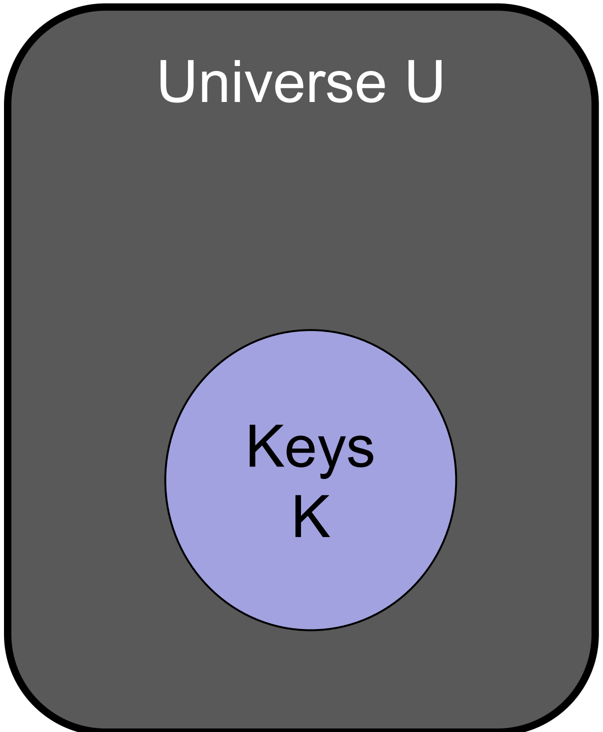
- Too much space
  - If keys are integers, then table-size > 4 billion
  - Hashing

Hash Functions

Problem:

e.g., 2^{140}

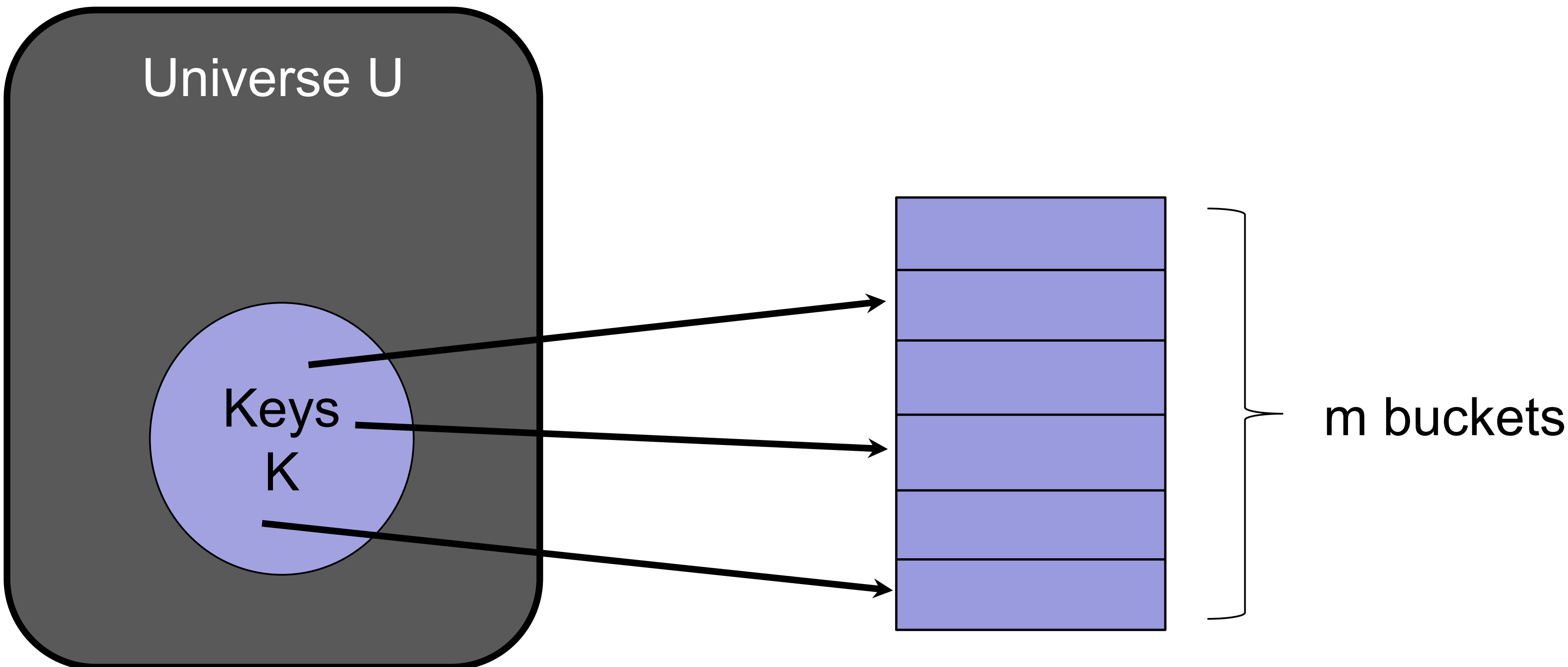
- Huge universe U of possible keys.
- Smaller number n of actual keys.



Hash Functions

Problem:

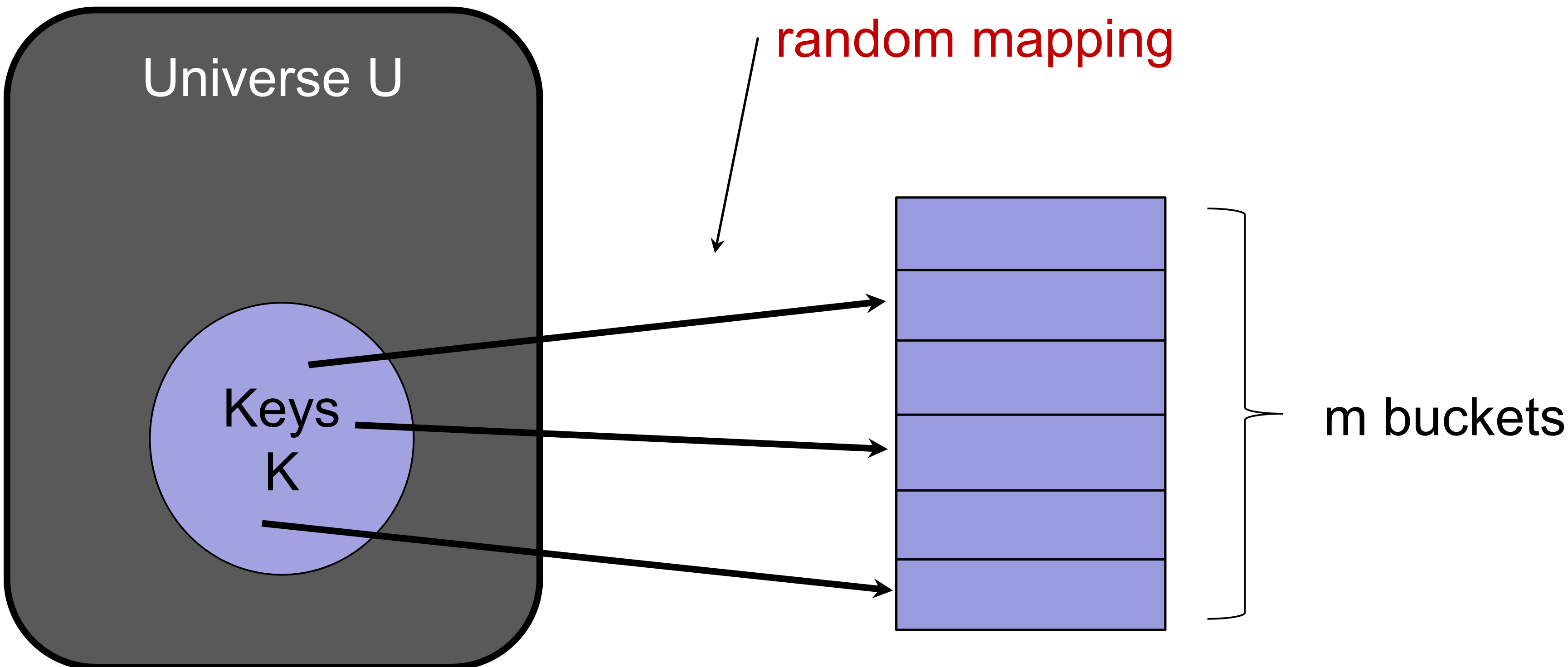
- Huge universe U of possible keys.
- Smaller number n of actual keys.
- How to map n keys to $m \approx n$ buckets?



Hash Functions

Problem:

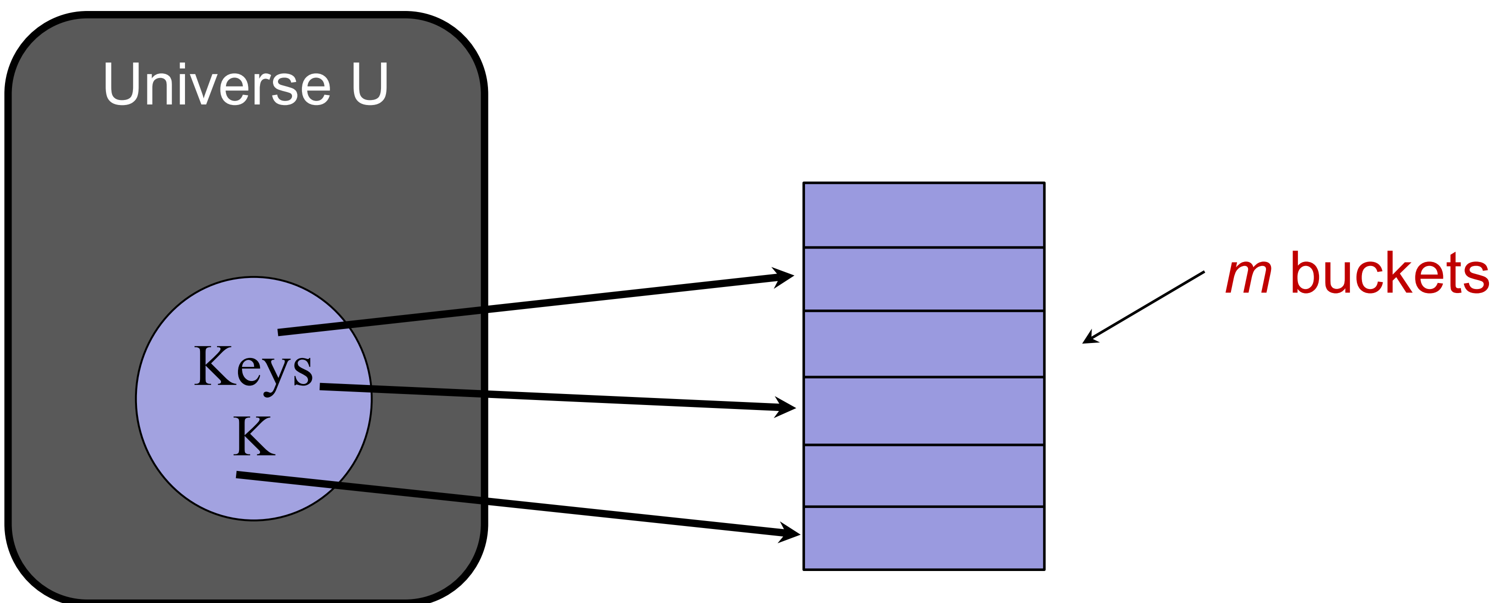
- Huge universe U of possible keys.
- Smaller number n of actual keys.
- How to map n keys to $m \approx n$ buckets?



Hash Functions

Define hash function $h : U \longrightarrow \{1..m\}$

- Store key k in bucket $h(k)$.
- Time complexity:
  - Time to compute h + Time to access bucket
- Usually: assume hash function has cost $O(1)$ to compute.

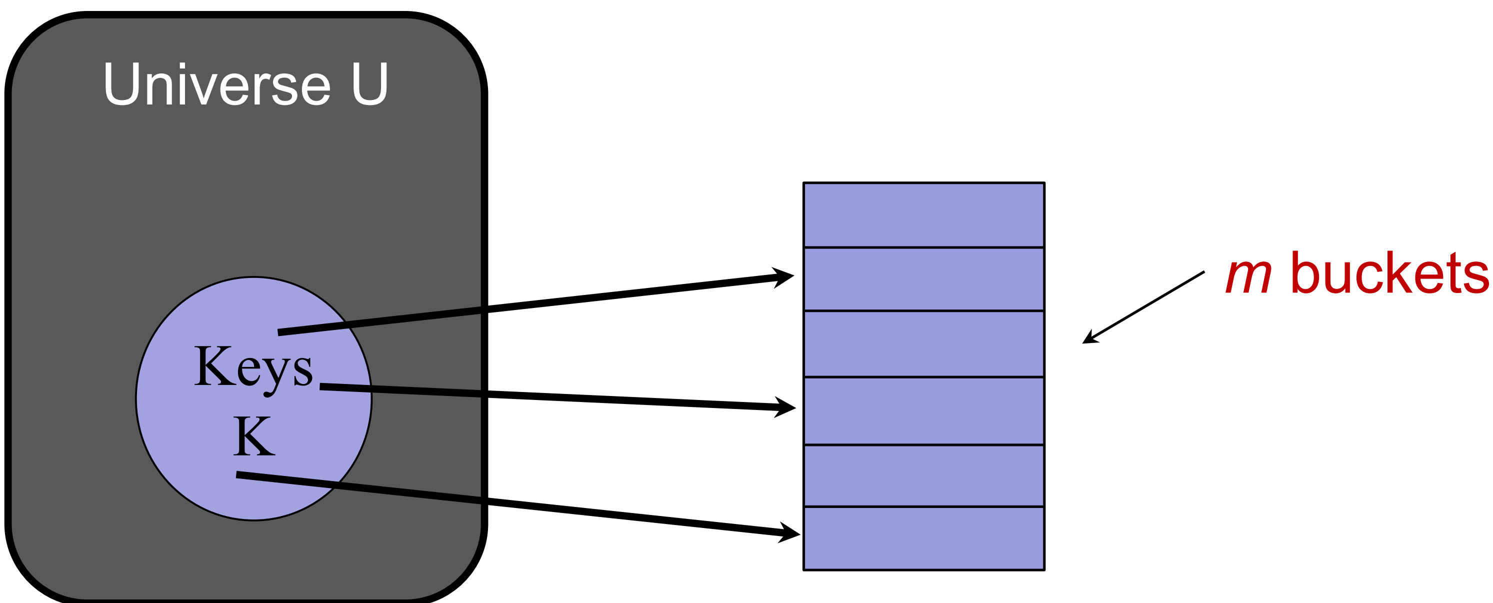


Hash Functions

Define hash function $h : U \longrightarrow \{1..m\}$

- Store key k in bucket $h(k)$.
- Time complexity:
  - Time to compute h + Time to access bucket
- Usually: assume hash function has cost O(1) to compute.

unless otherwise specified, e.g., long strings.



Hash Functions

Define hash function $h : U \longrightarrow \{1..m\}$

Think of a hash function as:

A function we "randomly" create before any insertions.

Hash Functions

Define hash function $h : U \longrightarrow \{1..m\}$

Think of a hash function as:

A function we "randomly" create before any insertions.

Then all insertions/lookups/deletions use the same function.

Hash Functions

Define hash function $h : U \longrightarrow \{1..m\}$

E.g. to hash numbers, maybe we pick a random prime p and value a, and then $h(x) = px + a$ is our hash function.

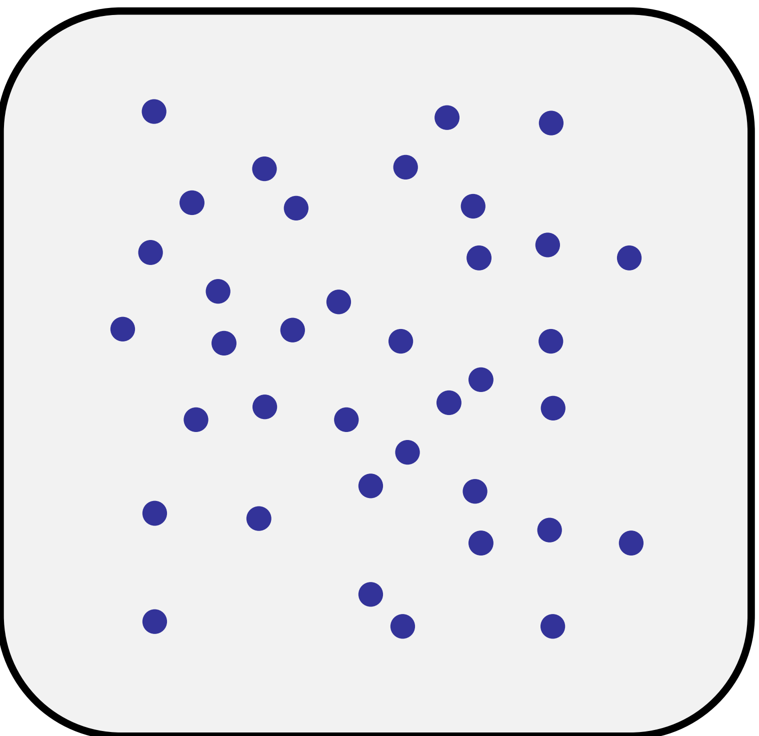
Hash Functions

Define hash function $h : U \longrightarrow \{1..m\}$

E.g. to hash numbers, maybe we pick a random prime p and value a, and then $h(x) = px + a$ is our hash function.

Then if our keys x to be inserted don't know what p and a are, they will be hashed to a random value, but given the same x, we always get the same value.

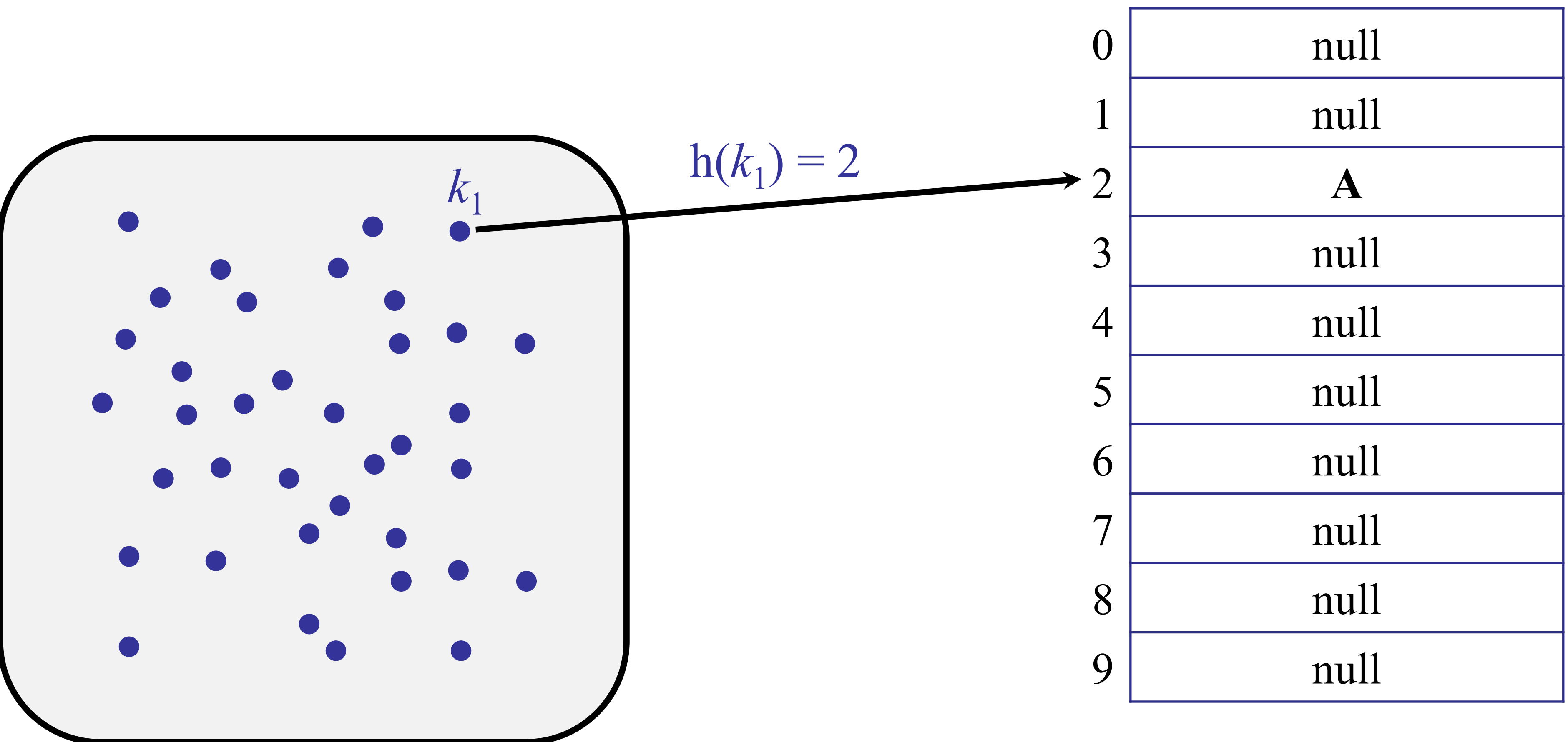
Hash Functions



0	null
1	null
2	null
3	null
4	null
5	null
6	null
7	null
8	null
9	null

Hash Functions

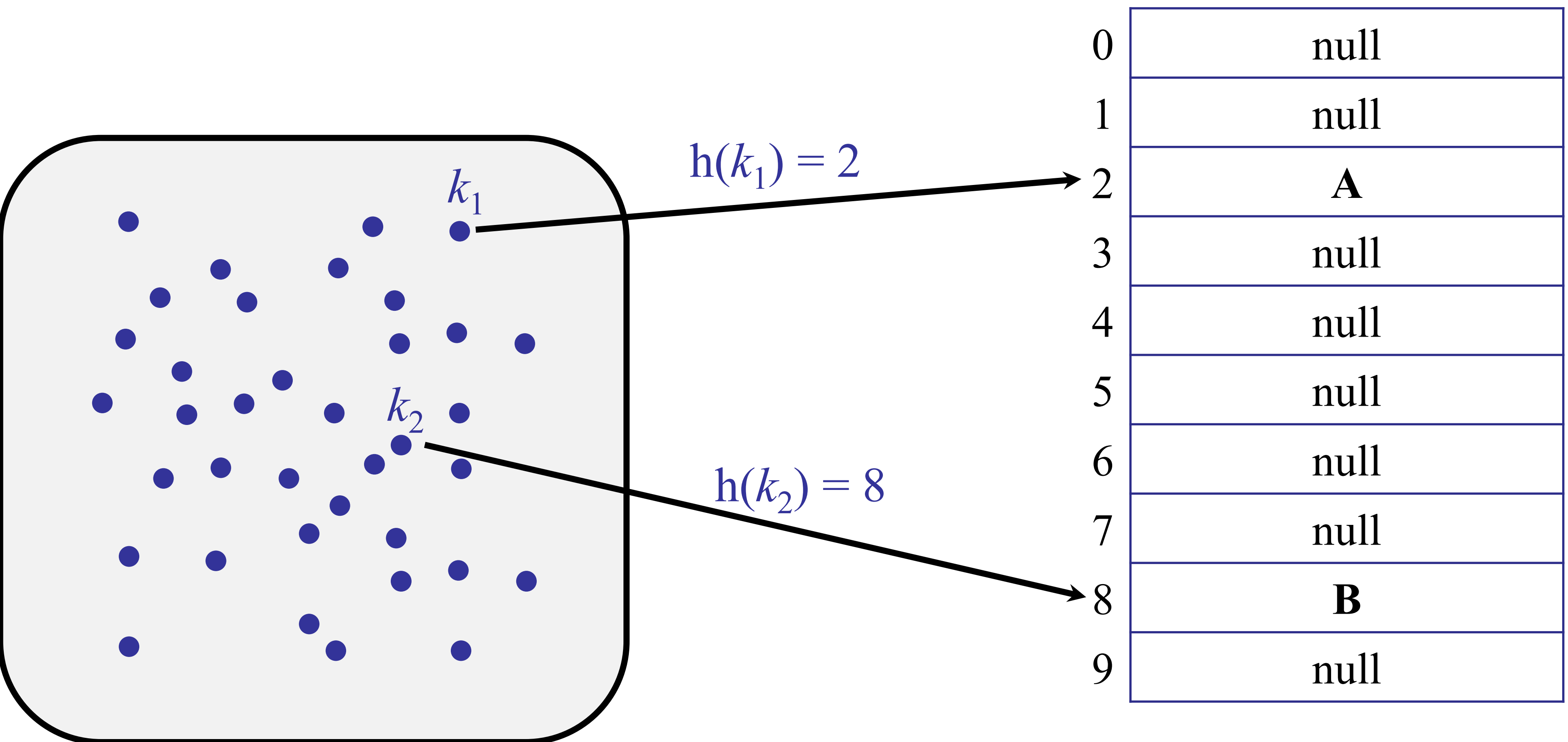
insert(k_1, A)



Hash Functions

insert(k_1, A)

insert(k_2, B)



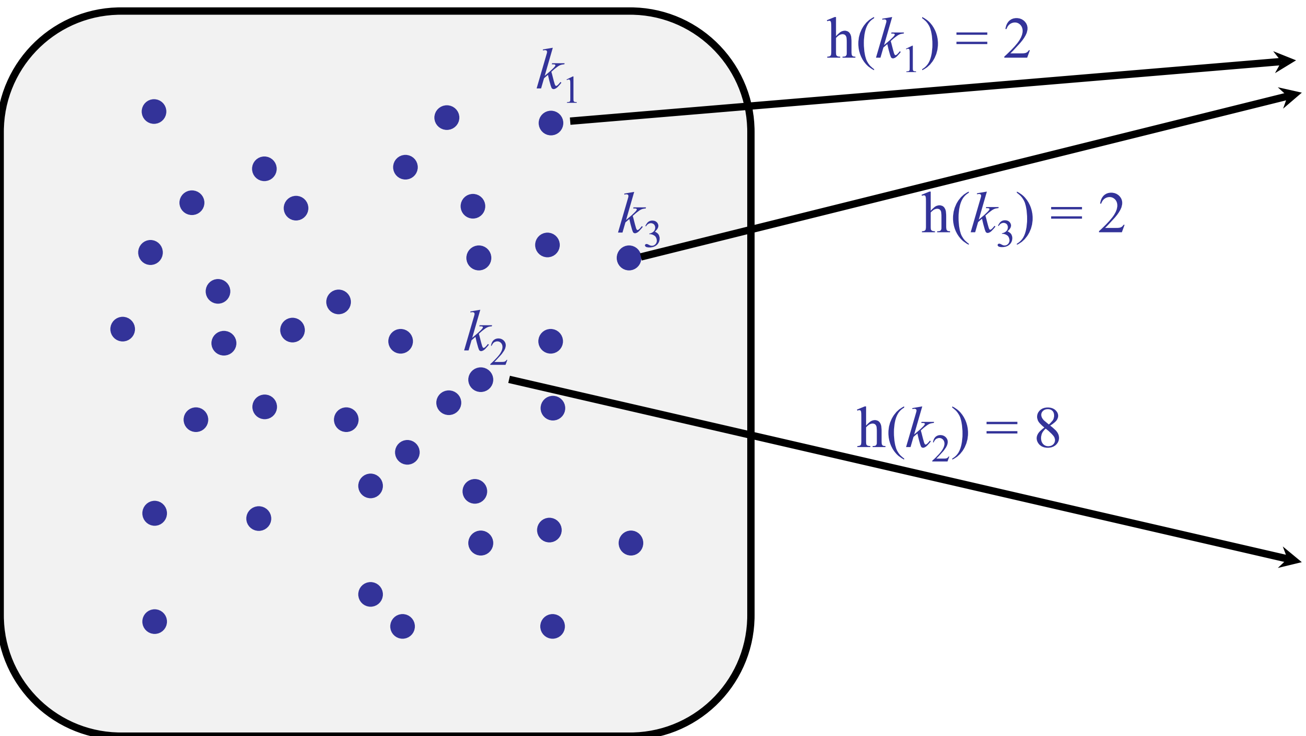
Hash Functions

insert(k_1, A)

insert(k_2, B)

insert(k_3, C)

Collision!



0	null
1	null
2	**A**
3	null
4	null
5	null
6	null
7	null
8	**B**
9	null

Hash Functions

Collisions:

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

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

Can we choose a hash function with no collisions?

1. Yes
2. Sometimes, if we choose carefully
3. No, impossible

Hash Functions

Collisions:

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

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

- 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: choose a new, better hash functions

Coping with Collision

Idea: choose a new, better hash functions

- Hard to find.
- Requires re-copying the table.
- Eventually, there will be another collision.

Coping with Collision

Idea: choose a new, better hash functions

- Hard to find.
- Requires re-copying the table.
- Eventually, there will be another collision.

If you don't and just change the hash function, we don't know how the keys were originally placed!

Coping with Collision

Idea: choose a new, better hash functions

- Hard to find.
- Requires re-copying the table.
- Eventually, there will be another collision.

Idea: chaining (today)

- Put both items in the same bucket!

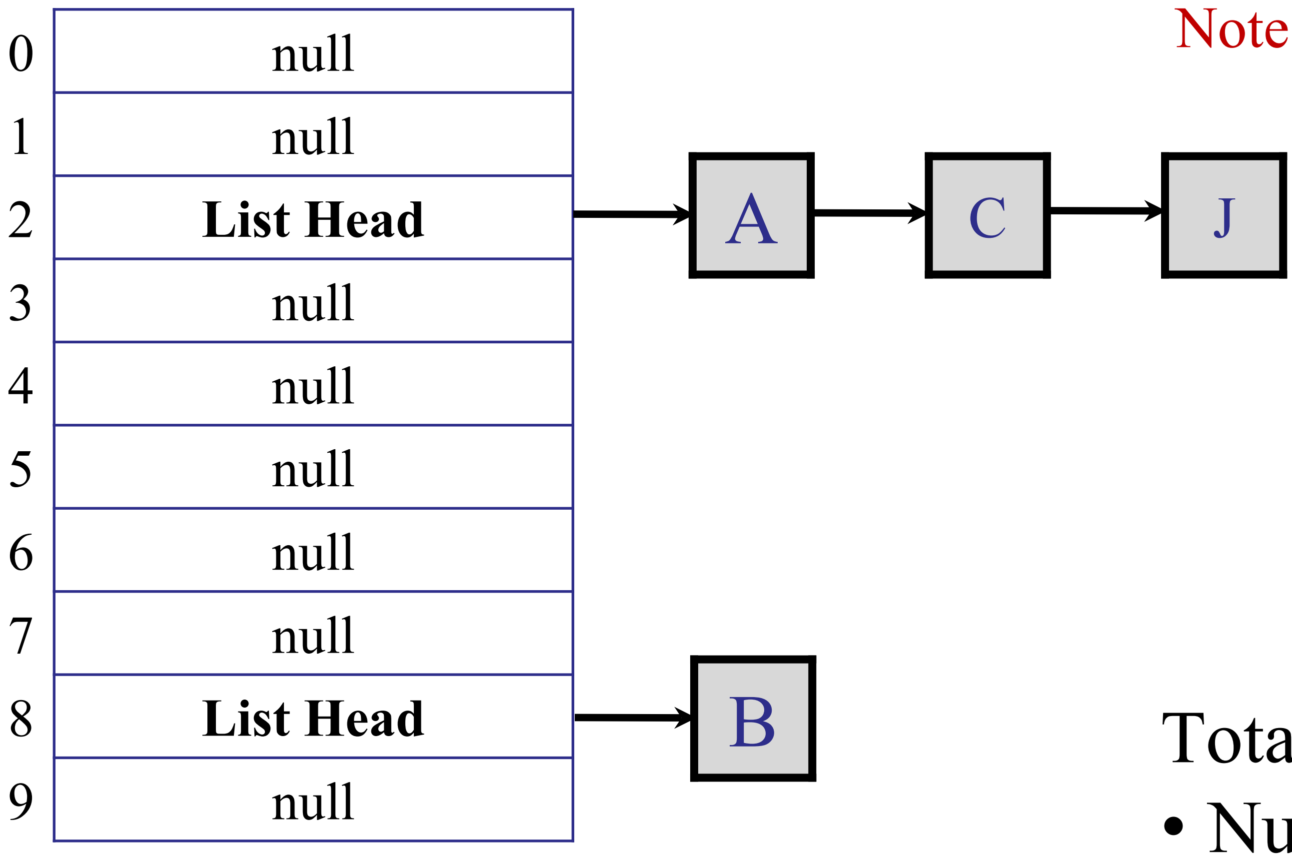
Idea: open addressing (next week)

- Find another bucket for the new item.

Chaining

Each bucket contains a linked list of items.

Note: h(A) == h(C) == h(J)

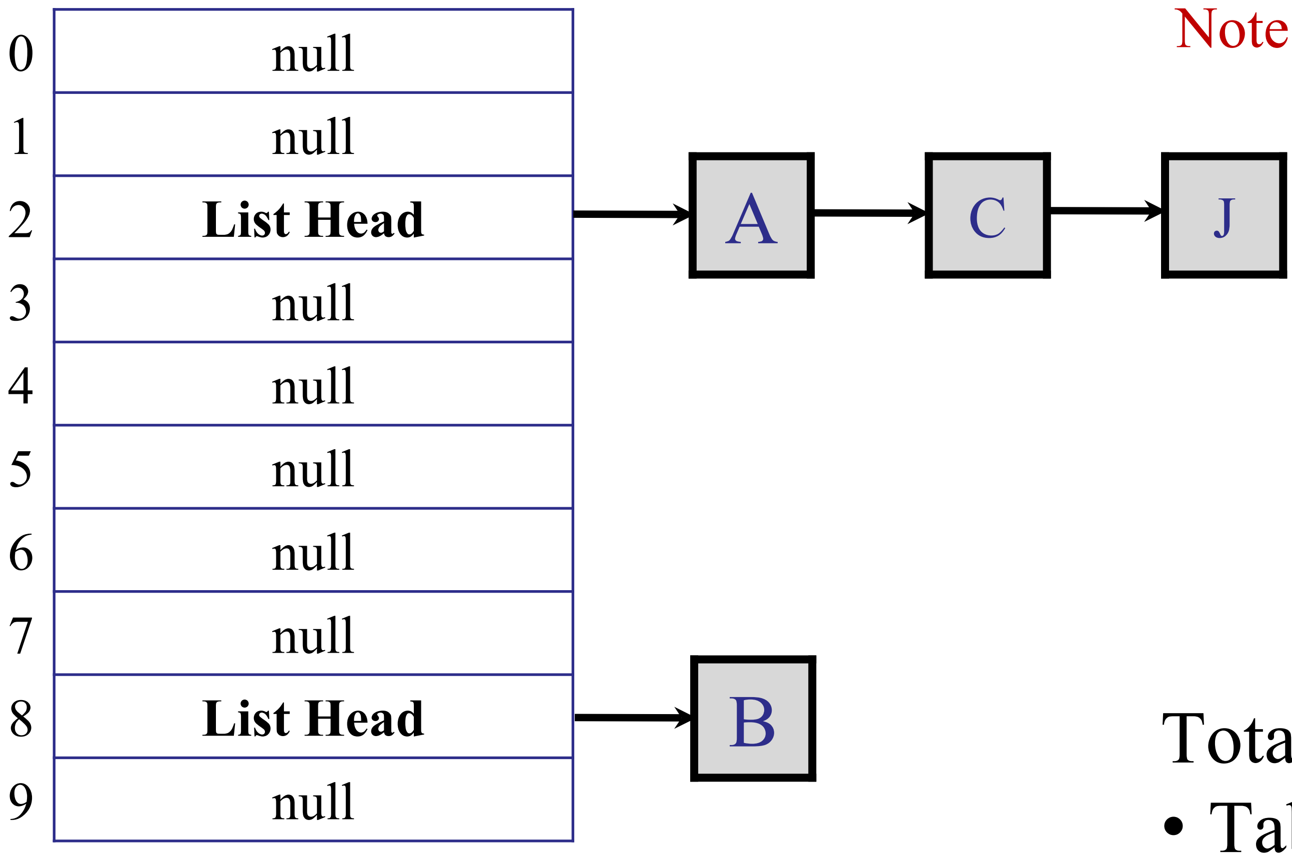


Total space:

- Number buckets: m
- Number entries: n

Chaining

Each bucket contains a linked list of items.



Note: h(A) == h(C) == h(J)

Total space: $O(m + n)$

- Table size: m
- Linked list size: n

Hashing with Chaining

Operations:

- insert(key, value)
  - Calculate h(key)
  - Lookup h(key) and add (key,value) to the linked list.

- search(key)
  - Calculate h(key)
  - Search for (key,value) in the linked list.

What is the worst-case cost of inserting a (key, value)? Assume cost(h) is cost of computing the hash function.

1. $O(1 + \text{cost}(h))$
2. $O(\log n + \text{cost}(h))$
3. $O(n + \text{cost}(h))$
4. $O(n \, \text{cost}(h))$
5. $O(n^2)$.

Do we care about duplicates?

- If so, the cost of insert is higher because we need to search for duplicates.

Hashing with Chaining

Operations:

Always O(1).

- insert(key, value)
  - Calculate h(key)
  - Lookup h(key) and add (key,value) to the linked list.

  (Note: this allows duplicate keys. Need to specify more precisely the behavior or insert!)

- search(key)
  - Calculate h(key)
  - Search for (key,value) in the linked list.

What is the worst-case cost of searching a (key, value)?

1. $O(1 + \text{cost}(h))$
2. $O(\log n + \text{cost}(h))$
3. $O(n + \text{cost}(h))$
4. $O(n * \text{cost}(h))$
5. We cannot determine it without knowing h.

Hashing with Chaining

Operations:

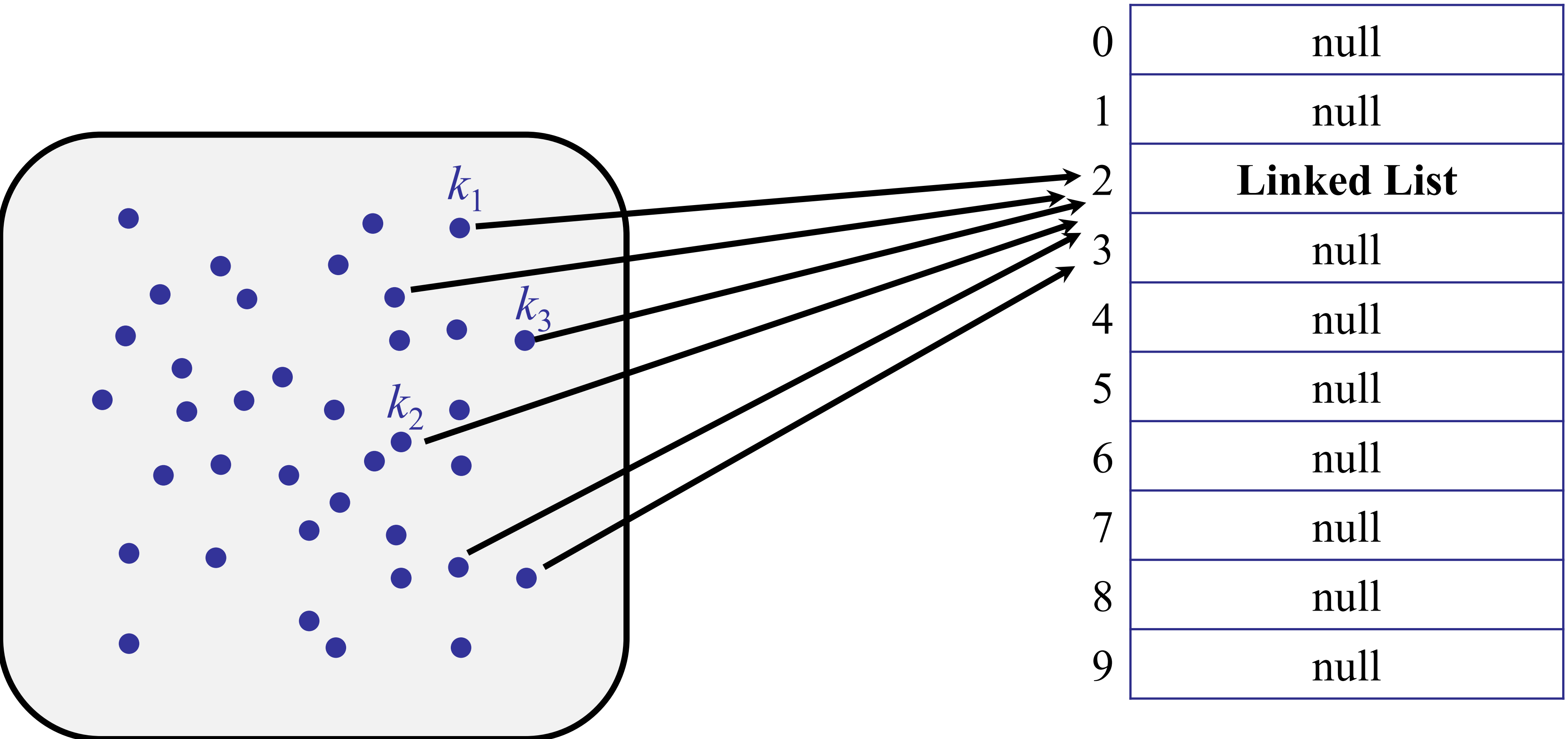
- insert(key, value)
  - Calculate h(key)
  - Lookup h(key) and add (key,value) to the linked list.

- search(key)   time depends on length of linked list
  - Calculate h(key)
  - Search for (key,value) in the linked list.

Hashing with Chaining

Assume all keys hash to the same bucket!

- Search costs O(n)
- Oh no!



Let's be optimistic today.

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.

Let's be optimistic today.

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.
- Importantly: the hash function always returns the same value on the same input. (It doesn't change over time!)

Why don't we just insert each key into a random bucket (instead of using a hash function h)?

Why don’t we just insert each key into a random bucket (instead of using hash function h)?

1. It would be slow to insert.
2. Computers don’t have a real source of randomness.
3. By choosing the keys carefully, a user could force the random choices to create many collisions.
4. Searching would be very slow.
5. None of the above.

Why don’t we just insert each key into a random bucket (instead of using hash function h)?

1. It would be slow to insert.
2. Computers don’t have a real source of randomness.
3. By choosing the keys carefully, a user could force the random choices to create many collisions.
4. Searching would be very slow.
5. None of the above.

Intuition: The hash function tells us where to find the key we are looking for. When we want to search the key again, we use the hash function again.

If we just randomly threw it into a bucket, we don’t know where we put it.

Let's be optimistic today.

The Simple Uniform Hashing Assumption

- Assume:
  - n items
  - m buckets
- Define: load(hash table) = n/m

  = (average # items) / bucket.

Want to show:

- Expected search time =1+(*expected # items per bucket)*

hash function + array access

linked list traversal

Probability Theory: Again

Set of outcomes for $X = (e_1, e_2, e_3, \ldots, e_k)$:

- $\Pr(e_1) = p_1$
- $\Pr(e_2) = p_2$
- …
- $\Pr(e_k) = p_k$

Expected outcome:

$$E[X] = e_1 p_1 + e_2 p_2 + \ldots + e_k p_k$$

Probability Theory: Again

Linearity of Expectation:

- $E[A + B] = E[A] + E[B]$

Example:

- A = # heads in 2 coin flips
- B = # heads in 2 coin flips
- A + B = # heads in 4 coin flips

$E[A+B] = E[A] + E[B] = 1 + 1 = 2$

Let's be optimistic today.

The Simple Uniform Hashing Assumption

- Assume:
  - n items
  - m buckets
- Define: load(hash table) = n/m

  = (average # items) / bucket.

Want to show:

- Expected search time =1+(*expected # items per bucket)*

hash function + array access

linked list traversal

A little more probability

Indicator random variables

$X(i, j) = 1$ if item i is put in bucket j

$= 0$ otherwise

A little more probability

Indicator random variables

$$X(i, j) = 1 \quad \text{if item } i \text{ is put in bucket } j$$
$$= 0 \quad \text{otherwise}$$

Recall:

The Simple Uniform Hashing Assumption

- Every key is equally likely to map to every bucket.
- Keys are mapped independently.

Pr(X(i, j) == 1) = ?

Probability that the ith item lands in bucket j? There are m possible buckets.

1. 1/m
2. 1/n
3. 1/(m+n)
4. m/n
5. n/m
6. log(n)

A little probability

Indicator random variables

$X(i, j) = 1$ if item i is put in bucket j

$= 0$ otherwise

$\mathbf{Pr}(X(i, j) == 1) = 1/m$

A little probability

Indicator random variables

$$X(i, j) = 1 \quad \text{if item } i \text{ is put in bucket } j$$
$$= 0 \quad \text{otherwise}$$

$$\mathbf{Pr}(X(i, j) == 1) = 1/m$$

$$\mathbf{E}(X(i, j)) = \mathbf{??}$$

A little probability

Indicator random variables

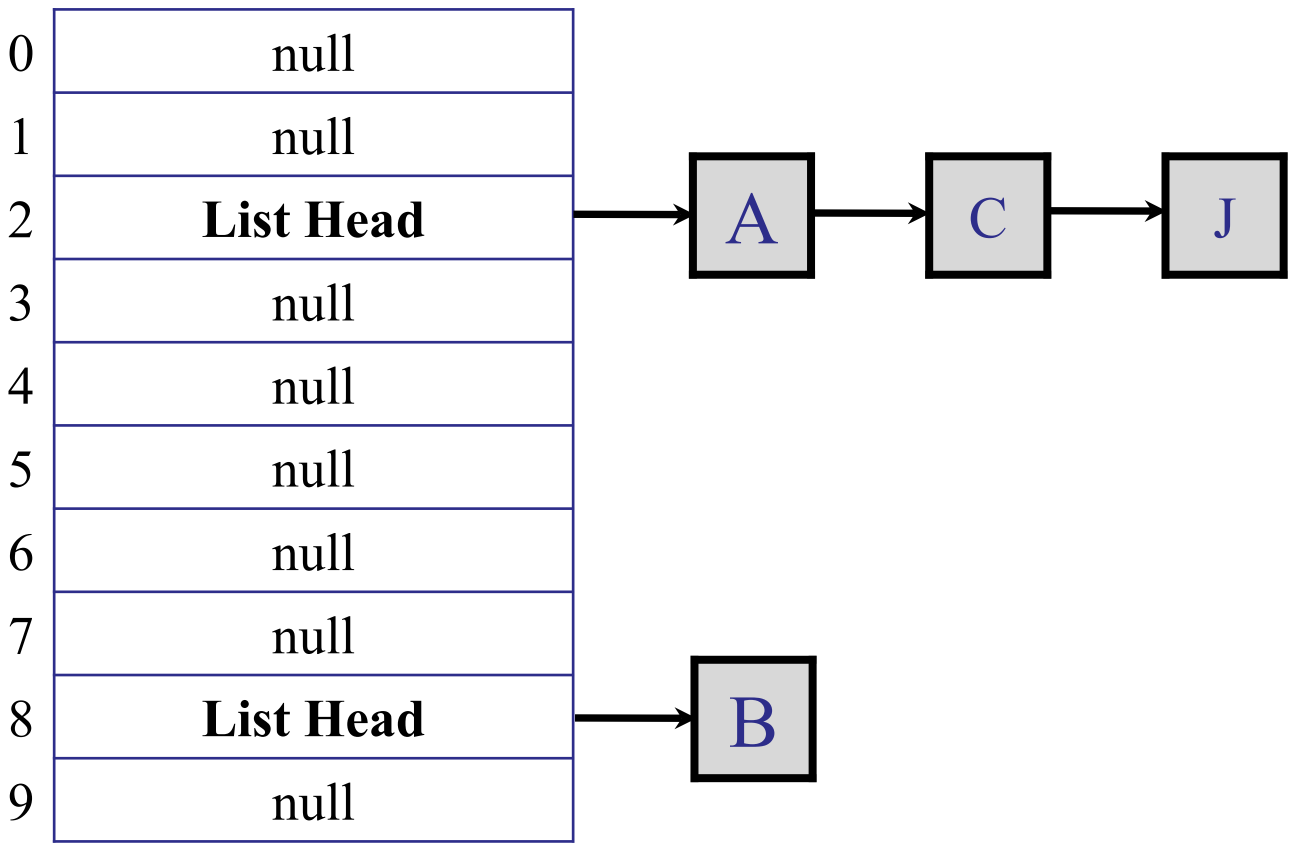
$$X(i, j) = 1 \quad \text{if item } i \text{ is put in bucket } j$$
$$= 0 \quad \text{otherwise}$$

$$\mathbf{Pr}(X(i, j) == 1) = 1/m$$

$$\mathbf{E}(X(i, j)) = \mathbf{Pr}(X(i, j) == 1) \times 1 + \mathbf{Pr}(X(i, j) == 0) \times 0$$
$$= \mathbf{Pr}(X(i, j) == 1)$$
$$= 1/m$$

A little probability

What is the expected number of items in a bucket?



A little probability

Indicator random variables

$X(i, j) = 1$ if item i is put in bucket j

$= 0$ otherwise

$\Sigma_i X(i, b) =$ number of items in bucket b

A little probability

Indicator random variables

$X(i, j) = 1$ if item i is put in bucket j

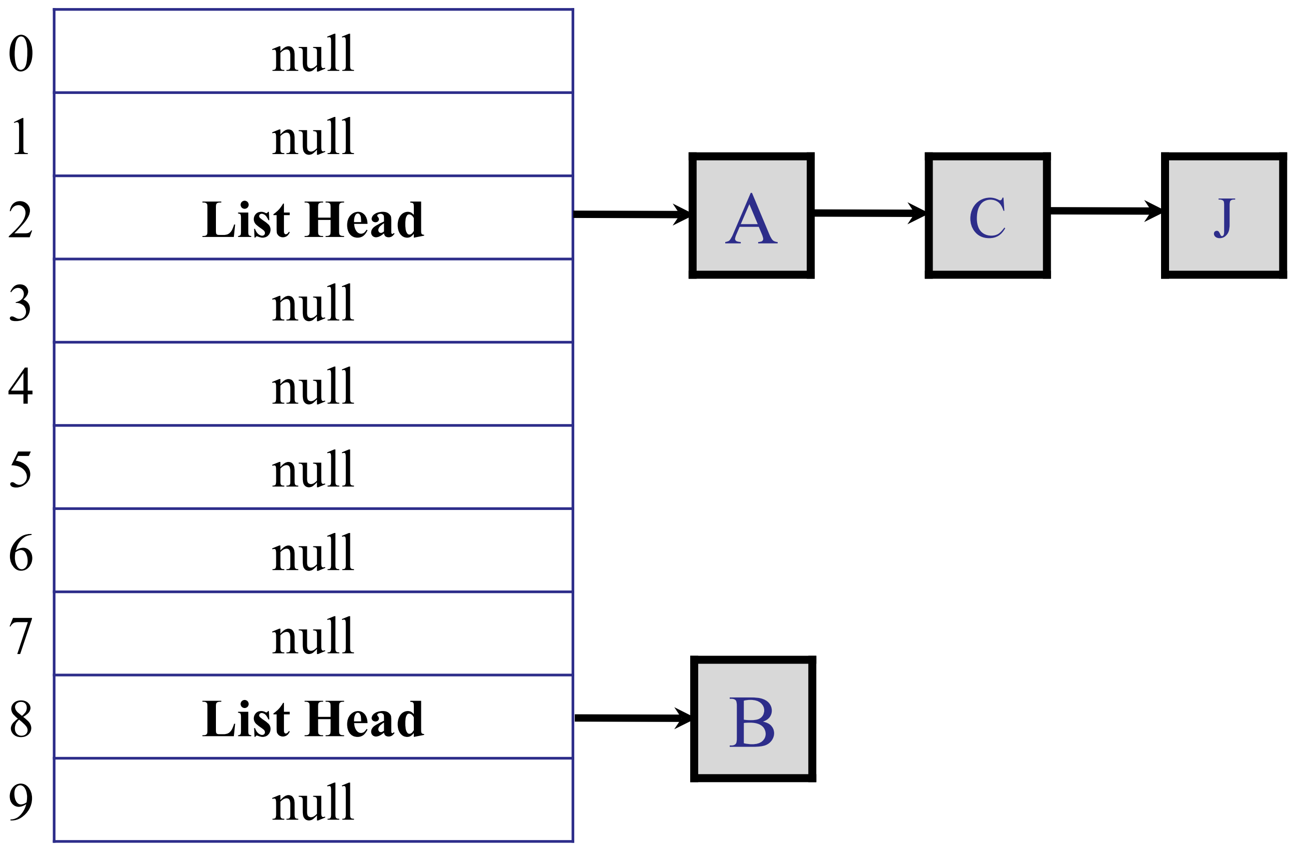
$= 0$ otherwise

$\Sigma_i X(i, b) =$ number of items in bucket b

Sum across all possible items. the items that hash to bucket b add 1 to the sum, otherwise it adds 0

A little probability

Each item contributes `1' to the bucket it is in..



A little probability

Calculate expected number of items per bucket:

Expected $(\Sigma_i X(i, b)) =$

A little probability

Calculate expected number of items per bucket:

$$\mathbf{E}(\Sigma_i X(i, b)) = \Sigma_i \mathbf{E}(X(i, b))$$

Linearity of expectation: $E(A + B) = E(A) + E(B)$

A little probability

Calculate expected number of items per bucket:

$$\mathbf{E}(\Sigma_i X(i, b)) = \Sigma_i \mathbf{E}(X(i, b))$$

$$= \Sigma_i 1/m$$

$$= n/m$$

Let's be optimistic today.

The Simple Uniform Hashing Assumption

- Assume:
  - n items
  - m buckets
- Define: load(hash table) = n/m

  = average # items / buckets.

Shown:

- Expected search time = $\underbrace{O(1)}_{\text{hash function + array access}} + \underbrace{n/m}_{\text{linked list traversal}}$

Let's be optimistic today.

The Simple Uniform Hashing Assumption

- Assume:
  - n items
  - $m = \Omega(n)$ buckets, e.g., $m = 2n$

We set the size of the table

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

$= O(1)$

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)

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)

Why not O(n)?

Hashing with Chaining

What if you insert n elements in your hash table?

What is the expected *maximum* cost?

Hashing with Chaining

What if you insert n elements in your hash table?

What is the expected *maximum* cost?

– Analogy:
  - Throw n balls in m = n bins.
  - What is the maximum number of balls in a bin?

Cost: $O(\log n)$

Hashing with Chaining

What if you insert n elements in your hash table?

What is the expected *maximum* cost?

– Analogy:
  - Throw n balls in m = n bins.
  - What is the maximum number of balls in a bin?

Cost: $\Theta(\log n / \log\log n)$

(See CS5330 for a proof.)

Hashing: Recap

Problem: coping with large universe of keys

- Number of possible keys is very, very large.
- Direct Access Table takes too much space

Hash functions

- Use hash function to map keys to buckets.
- Sometimes, keys collide (inevitably!)
- Use linked list to store multiple keys in one bucket.

Analyze performance with simple uniform hashing.

- Expected number of keys / bucket is $O(n/m) = O(1)$.

Summary

Symbol Tables are pervasive

- You find them everywhere!

Hash tables are fast, efficient symbol tables.

- Under optimistic assumptions, provably so.
- In the real world, often so.
- But be careful!

Beats BSTs:

- Operate directly on keys (i.e., indexing)
- Gave up: successor/predecessor/etc.