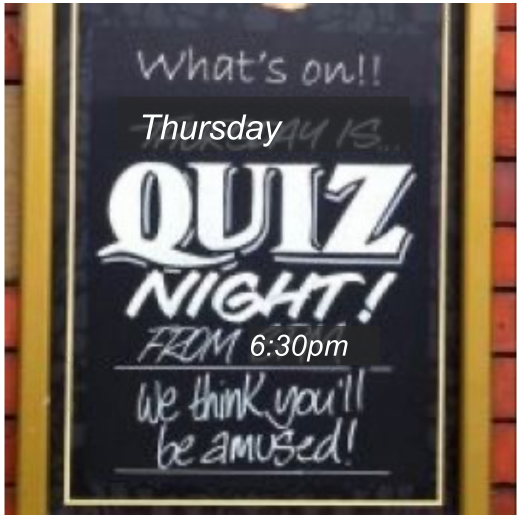
CS2040S
Data Structures and Algorithms

Hashing II

Today: More Hashing!

- Java hashing
- Collision resolution: open addressing
- Table (re)sizing

Midterm



Thurs. March 10 6:30pm

- Location: MPSH
- Room assignment on Coursemology
- 13+ rooms

Bring to quiz:

- One double-sided sheet of paper with any notes you like.
- Pens/pencils.
- You may not use anything else. (No calculators, no phones, etc.)

Today: More Hashing!

- Java hashing
- Collision resolution: open addressing
- Table (re)sizing

Review: Symbol Table Abstract Data Type

Which of the following is *not* typically a symbol table operation?

1. insert(key, data)
2. delete(key)
3. successor(key)
4. search(key)
5. None of the above.

ARCHIPELAGO

is open

Review: Symbol Table Abstract Data Type

Which of the following is *not* typically a symbol table operation?

1. insert(key, data)
2. delete(key)
3. successor(key)
4. search(key)
5. None of the above.

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.

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 $m = 10$.

(key, value)

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

Assume keys are distinct.

Direct Access Tables

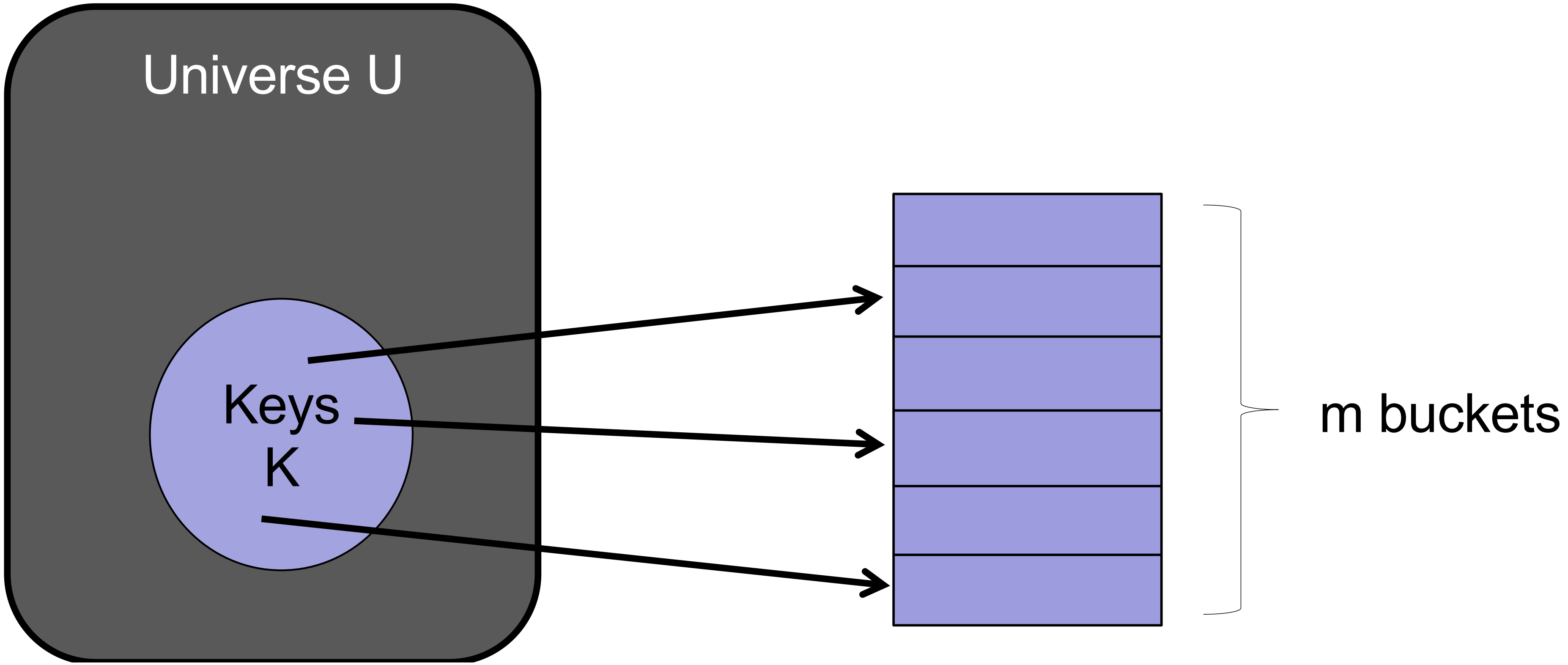
Problems:

- Too much space
  - If keys are 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…?

Hash Functions

Problem:

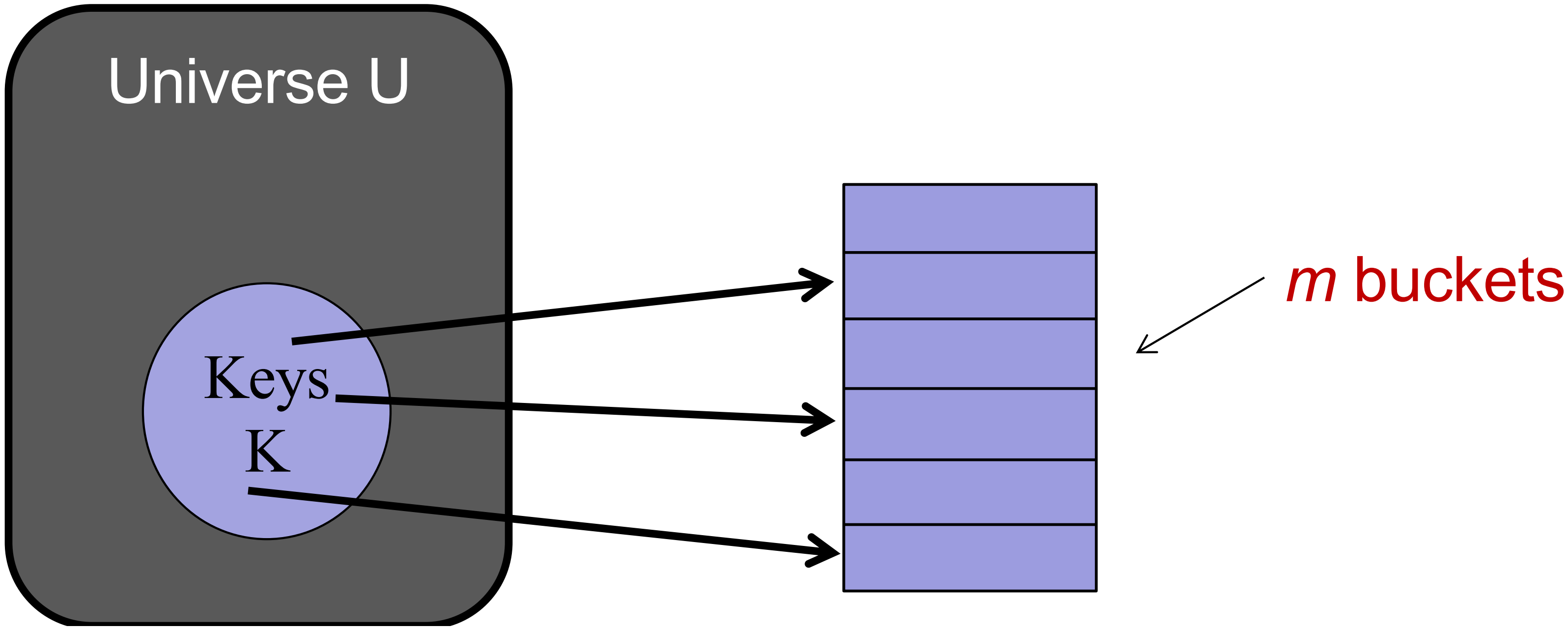
- Huge universe U of possible keys.
- Smaller number n of actual keys.
- How to map n keys to $m \approx n$ buckets?



Hash Functions

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

– Store key k in bucket $h(k)$.



Hash Functions

Collisions:

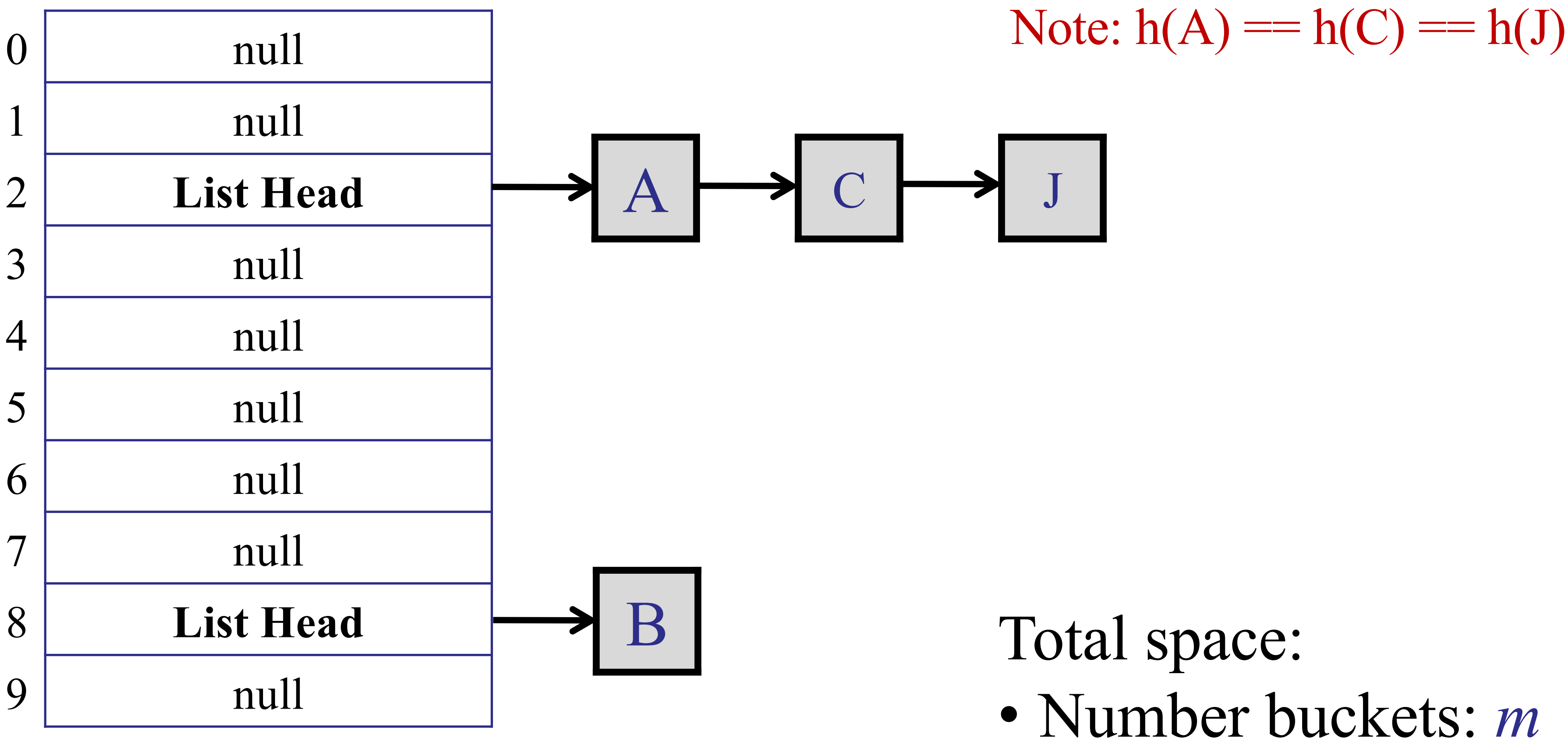
- We say that two distinct keys k_1 and k_2 collide if:

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

- 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!

Chaining

Each bucket contains a linked list of items.

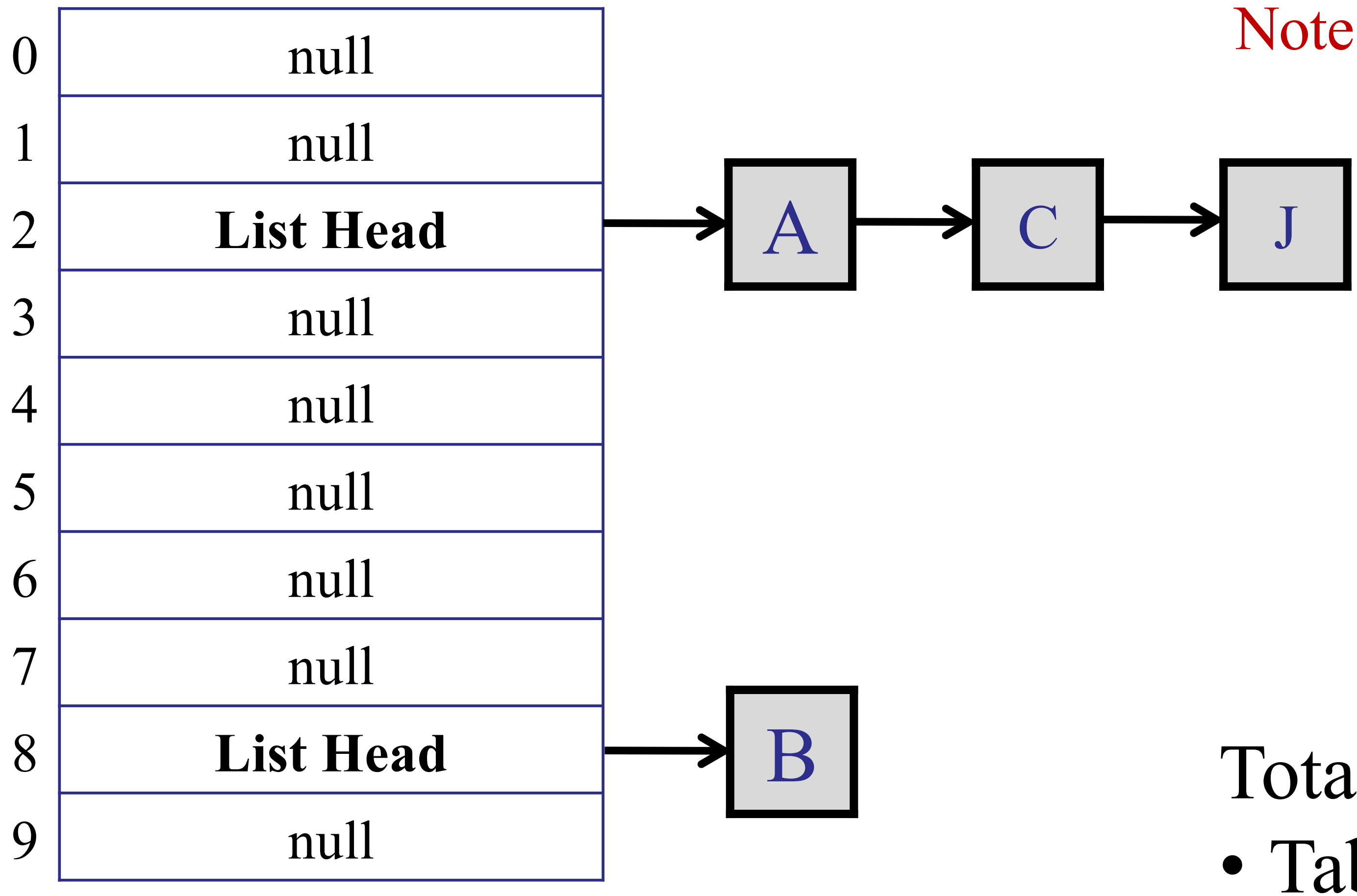


Total space:

- 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:

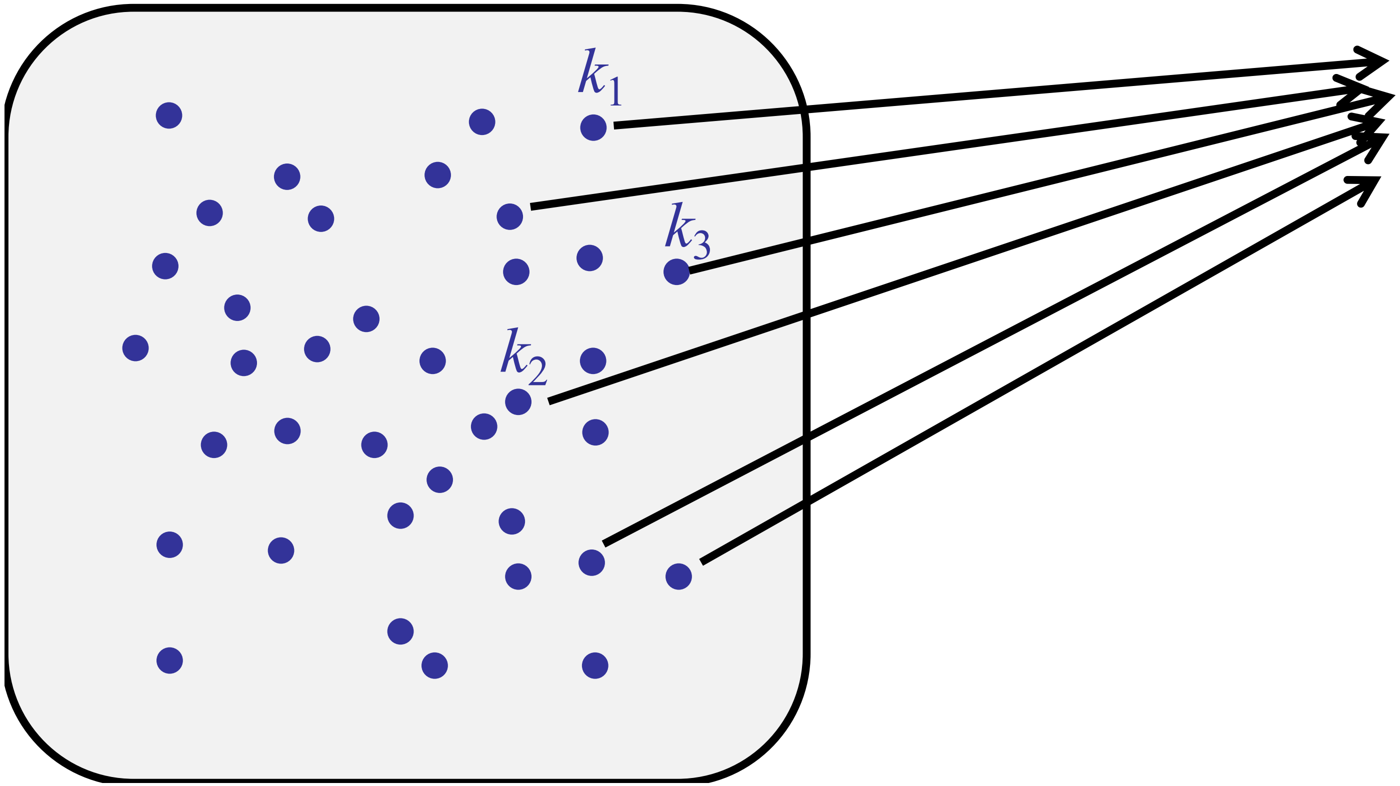
- insert(key, value)
  - Calculate h(key)
  - Lookup h(key) and add (key,value) to the linked list.

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

Hashing with Chaining

What if all keys hash to the same bucket!

- Worst-case search costs O(n)
- Oh no!



0	null
1	null
2	**Linked List**
3	null
4	null
5	null
6	null
7	null
8	null
9	null

Let's be optimistic today.

The Simple Uniform Hashing Assumption

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

Assume hash function has this property, even if it may not!

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.

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

Searching would be very slow. How do you find the item?

Review: Collisions

Assume m = table size, n = number of keys, and m=n. Assume simple uniform hashing assumption. Then what is the probability that two keys collide?

1. 1/2
2. 1/n
3. $1/n^2$
4. $1/n^3$
5. I don't understand

ARCHIPELAGO
is open

Let's be optimistic today.

The Simple Uniform Hashing Assumption

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

  = average # items / bucket.

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

hash function + array access

linked list traversal

A little probability

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

$\qquad\qquad = 0$ otherwise

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

Calculate expected number of items in bucket b:

$$\mathbf{E}(\Sigma_i X(i, b)) = \Sigma_i \mathbf{E}(X(i, b))$$
$$= \Sigma_i 1/m$$
$$= n/m$$

Let's be optimistic today.

The Simple Uniform Hashing Assumption

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

  = average # items / buckets.

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

Let's be optimistic today.

The Simple Uniform Hashing Assumption

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

- 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)

** In this case, inserting allows duplicates…
Preventing duplicates requires searching.

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 $O(n/m) = O(1)$.

Today

- Java hashing
- Collision resolution: open addressing
- Table (re)sizing

Symbol Tables in Java

Symbol Tables in Java

java.util.Map

public interface	**java.util.Map<Key, Value>**	
`void`	`clear()`	*removes all entries*
`boolean`	`containsKey(Object k)`	*is k in the map?*
`boolean`	`containsValue(Object v)`	*is v in the map?*
`Value`	`get(Object k)`	*get value for k*
`Value`	`put(Key k, Value v)`	*adds (k,v) to table*
`Value`	`remove(Object k)`	*remove mapping for k*
`int`	`size()`	*number of entries*

Note: no successor / predecessor queries.

Symbol Tables in Java

java.util.Map

Parameterized by key and value.
Not necessarily comparable

```
public interface  java.util.Map<Key, Value>

         void  clear()                     removes all entries
      boolean  containsKey(Object k)       is k in the map?
      boolean  containsValue(Object v)     is v in the map?
        Value  get(Object k)               get value for k
        Value  put(Key k, Value v)         adds (k,v) to table
        Value  remove(Object k)            remove mapping for k
          int  size()                      number of entries
```

Note: no successor / predecessor queries.

Symbol Tables in Java

java.util.Map

Search by key.

```
public interface  java.util.Map<Key, Value>

           void   clear()                     removes all entries
        boolean   containsKey(Object k)       is k in the map?
        boolean   containsValue(Object v)     is v in the map?
          Value   get(Object k)               get value for k
          Value   put(Key k, Value v)         adds (k,v) to table
          Value   remove(Object k)            remove mapping for k
            int   size()                      number of entries
```

Note: no successor / predecessor queries.

Symbol Tables in Java

java.util.Map

Search by value.
(May not be efficient.)

`public interface java.util.Map<Key, Value>`

`void`	`clear()`	*removes all entries*
`boolean`	`containsKey(Object k)`	*is k in the map?*
`boolean`	`containsValue(Object v)`	*is v in the map?*
`Value`	`get(Object k)`	*get value for k*
`Value`	`put(Key k, Value v)`	*adds (k,v) to table*
`Value`	`remove(Object k)`	*remove mapping for k*
`int`	`size()`	*number of entries*

Note: no successor / predecessor queries.

Symbol Tables in Java

java.util.Map

Can use any Object as key?

`public interface java.util.Map<Key, Value>`

`void`	`clear()`	*removes all entries*
`boolean`	`containsKey(Object k)`	*is k in the map?*
`boolean`	`containsValue(Object v)`	*is v in the map?*
`Value`	`get(Object k)`	*get value for k*
`Value`	`put(Key k, Value v)`	*adds (k,v) to table*
`Value`	`remove(Object k)`	*remove mapping for k*
`int`	`size()`	*number of entries*

Note: no successor / predecessor queries.

Symbol Tables in Java

java.util.Map

Put new (key, value) in table.

`public interface`	`java.util.Map<Key, Value>`	
`void`	`clear()`	*removes all entries*
`boolean`	`containsKey(Object k)`	*is k in the map?*
`boolean`	`containsValue(Object v)`	*is v in the map?*
`Value`	`get(Object k)`	*get value for k*
`Value`	`put(Key k, Value v)`	*adds (k,v) to table*
`Value`	`remove(Object k)`	*remove mapping for k*
`int`	`size()`	*number of entries*

Note: no successor / predecessor queries.

Map Interface in Java

java.util.Map<Key, Value>

- No duplicate keys allowed.
- No *mutable* keys

  If you use an *object* as a key, then you can't modify that object later.

Symbol Table

Key Mutability

```
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);
```

What time does this plane depart at?

Symbol Table

Key Mutability

Moral: Keys should be immutable.

Examples: Integer, String

```
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);
```

Design Decisions

Allow duplicate keys?

- No: need to search on insertion
- Yes: faster insertion

What to do if user inserts duplicate key?

- Replace existing key.
- Add new value (i.e., key has two values).
- Error.

Insert empty/null value?

- Deletes existing (key, value) pair.
- Creates a null value.
- Error.

Symbol Tables in Java

java.util.Map

	`public interface`	`java.util.Map<Key, Value>`
`Set<Map.Entry<Key, Value>`	`entrySet()`	*set of all mappings*
`Set<Key>`	`keySet()`	*set of all keys*
`Collection<Value>`	`values()`	*collection of all values*

Note: not sorted

not necessarily efficient to work with these sets/collections.

What is wrong here?

Example:

There is a bug here!

```
Map<String, Integer> ageMap = new Map<String, Integer>();

ageMap.put("Alice", 32);
ageMap.put("Bernice", 84);
ageMap.put("Charlie", 7);

Integer age = ageMap.get("Alice")
```

- Key-type: String
- Value-type: Integer

ARCHIPELAGO
is open

What is wrong here?

Example:

Map is an interface!
Cannot instantiate an interface.

```
Map<String, Integer> ageMap = new Map<String, Integer>();

ageMap.put(“Alice”, 32);
ageMap.put(“Bernice”, 84);
ageMap.put(“Charlie”, 7);

Integer age = ageMap.get(“Alice”)
```

- Key-type: String
- Value-type: Integer

Map Class in Java

Example: HashMap

```
Map<String, Integer> ageMap = new HashMap<String, Integer>();

ageMap.put("Alice", 32);
ageMap.put("Bernice", 84);
ageMap.put("Charlie", 7);

Integer age = ageMap.get("Alice");
System.out.println("Alice's age is: " + age + ".");
```

- Key-type: String
- Value-type: Integer

Map Class in Java

Example: HashMap

```
Map<String, Integer> ageMap = new HashMap<String, Integer>();

ageMap.put("Alice", 32);
ageMap.put("Bernice", null);
ageMap.put("Charlie", 7);

Integer age = ageMap.get("Bob");
if (age==null){
    System.out.println("Bob's age is unknown.");
}
```

- Returns “null” when key is not in map.
- Returns “null” when value is null.

Map Classes in Java

HashMap

Symbol Table

- containsKey
- containsValue
- entrySet
- get
- isEmpty
- keySet
- put
- putAll
- remove
- values

TreeMap

Dictionary

- containsKey
- containsValue
- entrySet
- get
- isEmpty
- keySet
- put
- putAll
- remove
- values

Map Classes in Java

HashMap

Symbol Table

TreeMap

Dictionary

- ceilingEntry
- ceilingKey
- descendingKeySet
- firstEntry
- firstKey
- floorEntry
- floorKey
- headMap
- higherEntry
- higherKey
- … (and more)

Lots of functionality

Wide Interfaces

vs.

Narrow Interfaces

Limited functionality

Lots of functionality
- Java
- No guarantee of efficiency.
- Easy to use (badly).

Wide Interfaces

vs.

Narrow Interfaces

Limited functionality
- Enforces proper use.
- Restricts usage.

Which do you prefer?

Narrow Wide

Yes or No on Zoom.

Hashing in Java

How does your program know which hash function to use?

```
HashMap<MyFoo, Integer> hmap = new ...
MyFoo foo = new MyFoo();

hmap.put(foo, 8);
```

Java Hash Functions

Every object supports the method:

```
int hashCode()
```

Java Object

Every class implicitly extends Object

`public class Object`

`Object`	`clone()`	*creates a copy*
`boolean`	**`equals(Object obj)`**	***is obj equal to this?***
`void`	`finalize()`	*used by garbage collector*
`Class`	`getClass()`	*returns class*
`int`	**`hashCode()`**	***calculates hash code***
`void`	`notify()`	*wakes up a waiting thread*
`void`	`notifyAll()`	*wakes up all waiting threads*
`String`	**`toString()`**	***returns string representation***
`void`	`wait(…)`	*wait until notified*

Hashing in Java

How does your program know which hash function to use?

```
HashMap<MyFoo, Integer> hmap = new ...
MyFoo foo = new MyFoo();

int hash = foo.hashCode();

hmap.put(foo, 8);
```

Java Hash Functions

Every object supports the method:

```
int hashCode()
```

Rules:

- Always returns the same value, if the object hasn't changed.
- If two objects are equal, then they return the same hashCode.

No random hashcodes!

Is it legal for every object to return 32?

Java Hash Functions

Every object supports the method:

```
int hashCode()
```

Rules:

- Always returns the same value, if the object hasn't changed.
- If two objects are equal, then they return the same hashCode.

Is it *legal* for every object to return 32? (YES)

Java Hash Functions

Every object supports the method:

```
int hashCode()
```

Default Java implementation:

- hashCode returns the memory location of the object
- Every object hashes to a different location

Must implement/override `hashCode()` for your class.

Java Library Classes

`Integer`

`Long`

`String`

Integer

```
public int hashCode() {
   return value;
}
```

Rules:
- Always returns the same value, if the object hasn't changed.
- If two objects are equal, then they return the same hashCode.

Note: hashcode is always a 32-bit integer.

Note: every 32-bit integer gets a unique hashcode.

What do you do for smaller hash tables?
Can there be collisions?

Long

Collision can happen!

```
public int hashCode() {
   return (int)(value ^ (value >>> 32));
}
```

$$\text{hash}(\underbrace{0\,1\,1\,0\,0\,1\,0\,1\,1\,0\,0\,1\,1\,1\,0\,0}_{\text{32 bits}}\,\underbrace{0\,0\,1\,0\,1\,0\,0\,0\,0\,1\,1\,0\,0\,1\,0\,0}_{\text{32 bits}})$$

	0 1 1 0 0 1 0 1 1 0 0 1 1 1 0 0
XOR	0 0 1 0 1 0 0 0 0 1 1 0 0 1 0 0
	0 1 0 0 1 1 0 1 1 1 1 1 1 0 0 0

String

```
public int hashCode() {
   int h = hash; // only calculate hash once
   if (h == 0 && count > 0) {  // empty = 0
        int off = offset;
        char val[] = value;
        int len = count;
        for (int i = 0; i < len; i++) {
             h = 31*h + val[off++];
        }
        hash = h;
   }
   return h;
}
```

String

HashCode calculation:

```
hash = s[0]*31^(n-1) +
       s[1]*31^(n-2) +
       s[2]*31^(n-3) +
       ... +
       s[n-2]*31 +
       s[n-1]
```

$$hash = s[0]*31^{(n-1)} + s[1]*31^{(n-2)} + s[2]*31^{(n-3)} + \dots + s[n-2]*31 + s[n-1]$$

Why did they choose 31?

String

HashCode calculation:

```
hash = s[0]*31^(n-1) +
       s[1]*31^(n-2) +
       s[2]*31^(n-3) +
       ... +
       s[n-2]*31 +
       s[n-1]
```

Why did they choose 31? Prime, 2^5-1

Creating a new class

```
public class Pair {
  private int first;
  private int second;

  Pair(int a, int b){
      first = a;
      second = b;
  }
}
```

Creating a new class

```
public void testPair() {

  HashMap<Pair, Integer> htable =
          new HashMap<Pair, Integer>();

  Pair one = new Pair(20, 40);
  htable.put(one, 7);

  Pair two = new Pair(20, 40);
  int question = htable.get(two);
}
```

htable.get(new Pair(20, 40)) == ?

1. 1
2. 7
3. 11
4. null

ARCHIPELAGO

is open

Creating a new class

```
Pair one = new Pair(20, 40);
Pair two = new Pair(20, 40);

one.hashCode() != two.hashCode()
```

Creating a new class

```
Pair one = new Pair(20, 40);
Pair two = new Pair(20, 40);
htable.put(one, "first item");

htable.get(one) ➔ "first item"

htable.get(two) ➔ null
```

Creating a new class

```
public class Pair {
  private int first;
  private int second;

  Pair(int a, int b){
     first = a;
     second = b;
  }

  int hashCode(){
     return (first ^ second);
  }
}
```

Creating a new class

```
Pair one = new Pair(20, 40);
Pair two = new Pair(20, 40);
htable.put(one, "first item");

htable.get(one) → "first item"
htable.get(two) → null

one.equals(two) → false
```

Java Hash Functions

Every object supports the method:

```
int hashCode()
```

Rules:

- Always returns the same value, if the object hasn’t changed.
- If two objects are equal, then they return the same hashCode.
- **Must redefine .equals to be consistent with hashCode.**

Creating a new class

```
Pair one = new Pair(20, 20);
Pair two = new Pair(20, 20);
htable.put(one, “first item”);

htable.get(one) => “first item”

htable.get(two) => null
```

Java Hash Functions

Every object supports the method:

```
boolean equals(Object o)
```

Rules:

- **Reflexive**: x.equals(x) ➔ true
- **Symmetric**: x.equals(y) == y.equals(x)
- **Transitive**: x.equals(y), y.equals(z) ➔ x.equals(z)
- **Consistent**: always returns the same answer
- **Null is null**: x.equals(null) ➔ false

Java Hash Functions

Every object supports the method:

```
boolean equals(Object o)
```

```
boolean equals(Object p){
  if (p == null) return false;
  if (p == this) return true;

  if (!(p instanceOf Pair)) return false;
  Pair pair = (Pair)p;

  if (pair.first != first) return false;
  if (pair.second != second) return false;
  return true;
}
```

Java HashMap

```
public V get(Object key) {
   if (key == null) return getForNullKey();
   int hash = hash(key.hashCode());
   for (Entry<K,V> e = table[indexFor(hash,table.length)];
        e != null;
        e = e.next)
   {
      Object k;
      if (e.hash==hash &&((k=e.key)==key)||key.equals(k)))
         return e.value;
   }
   return null;
}
```

Java HashMap

```java
// This function ensures that hashCodes that differ only
// by constant multiples at each bit position have a
// bounded number of collisions (approximately 8 at
// default load factor).

static int hash(int h) {
   h ^= (h >>> 20) ^ (h >>> 12);
   return h ^ (h >>> 7) ^ (h >>> 4);
}
```

Java HashMap

```
public V get(Object key) {
   if (key == null) return getForNullKey();
   int hash = hash(key.hashCode());
   for (Entry<K,V> e = table[indexFor(hash,table.length)];
        e != null;
        e = e.next)
   {
      Object k;
      if (e.hash==hash &&((k=e.key)==key)||key.equals(k)))
         return e.value;
   }
   return null;
}
```

Java HashMap

```
public V get(Object key) {
   if (key == null) return getForNullKey();
   int hash = hash(key.hashCode());
   for (Entry<K,V> e = table[indexFor(hash,table.length)];
        e != null;
        e = e.next)
   {
      Object k;
      if (e.hash==hash &&((k=e.key)==key)||key.equals(k)))
         return e.value;
   }
   return null;
}
```

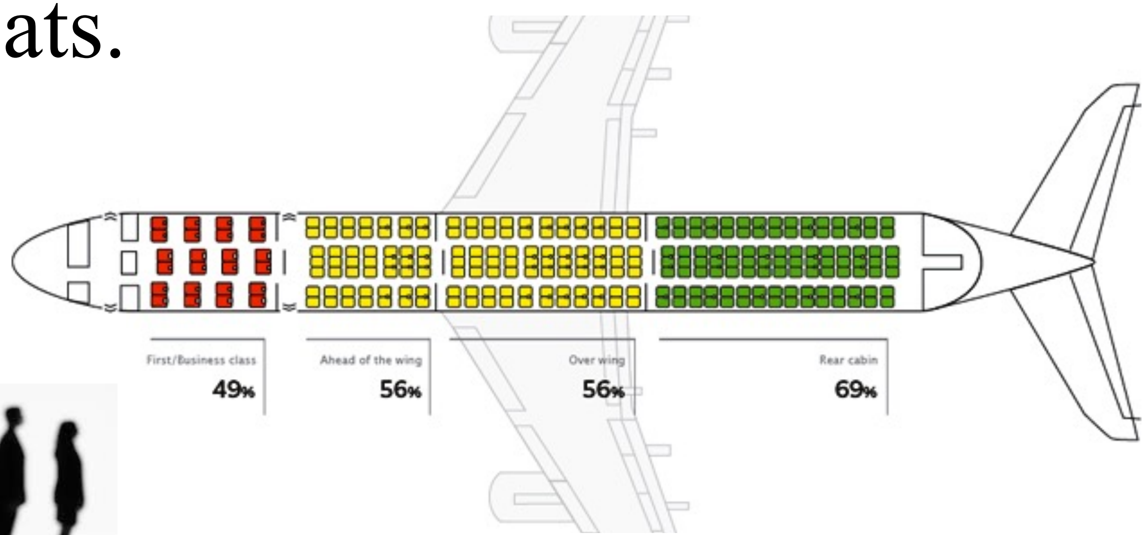
Java checks if the key is equal to the item in the hash table before returning it!

Today

- Java hashing
- Collision resolution: open addressing
- Table (re)sizing

Puzzle Break

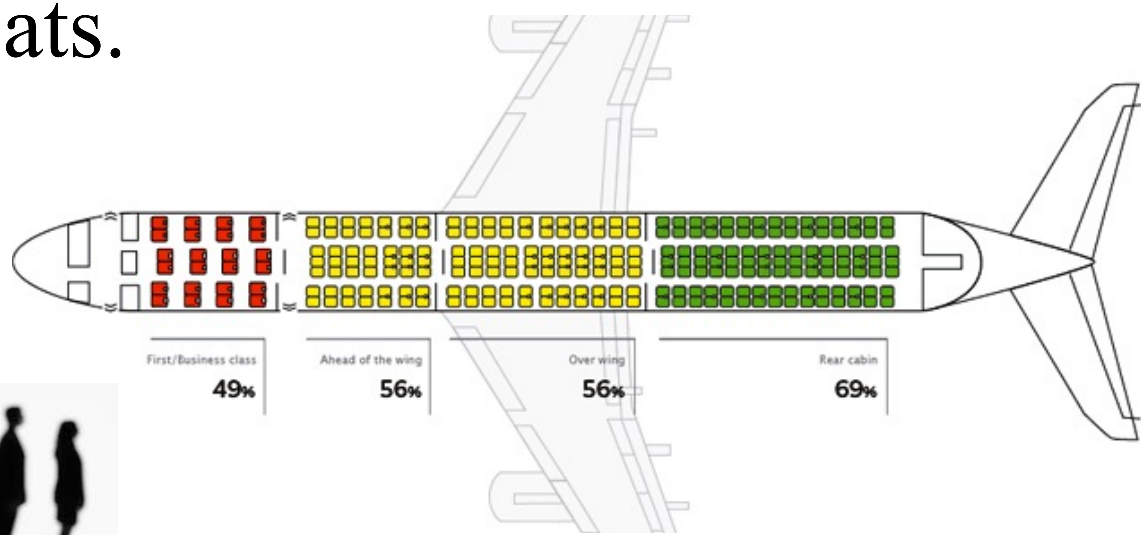
An airplane has 100 seats.



100 passengers board the airplane in a random order.

Puzzle Break

An airplane has 100 seats.



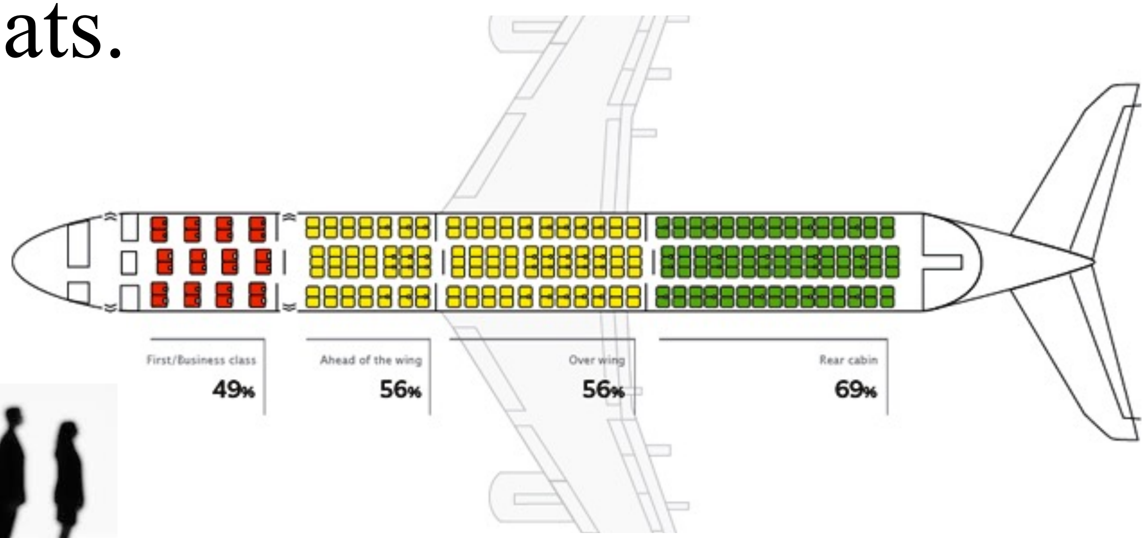
100 passengers board the airplane in a random order.

Passenger 1 is Mr. Burns.

Mr. Burns sits in a random seat.

Puzzle Break

An airplane has 100 seats.



Every other passenger:

- Sits in their assigned seat, if it is free.
- Otherwise, sits in a random seat.

Puzzle Break

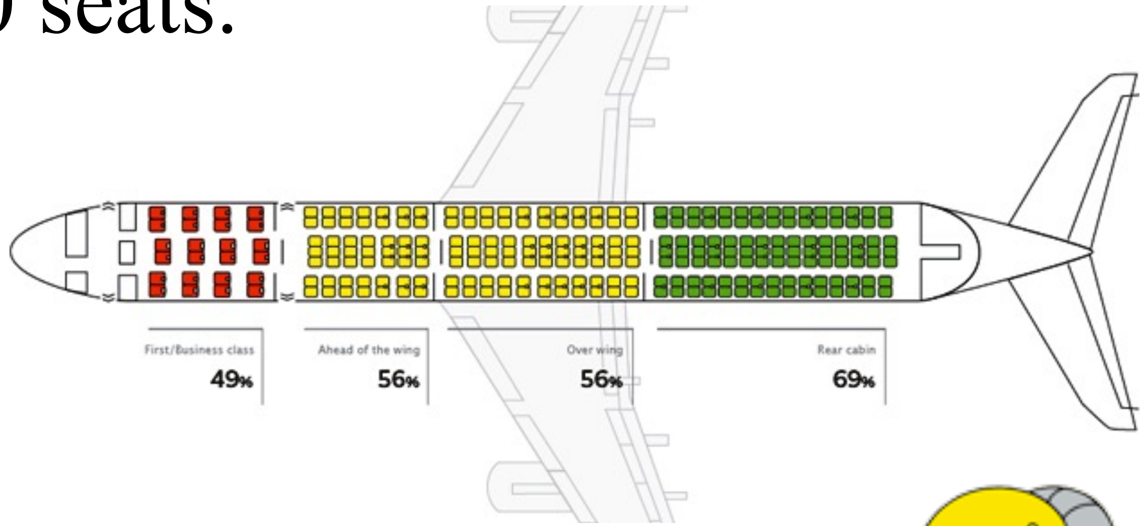
An airplane has 100 seats.

You are passenger #100.

What is the probability your seat is free when you board?

Puzzle Break

An airplane has 100 seats.



What is the probability your seat is free when you board?

Problem Solving techniques:

Try a plane with 2 seats. Try a plane with 3 seats.

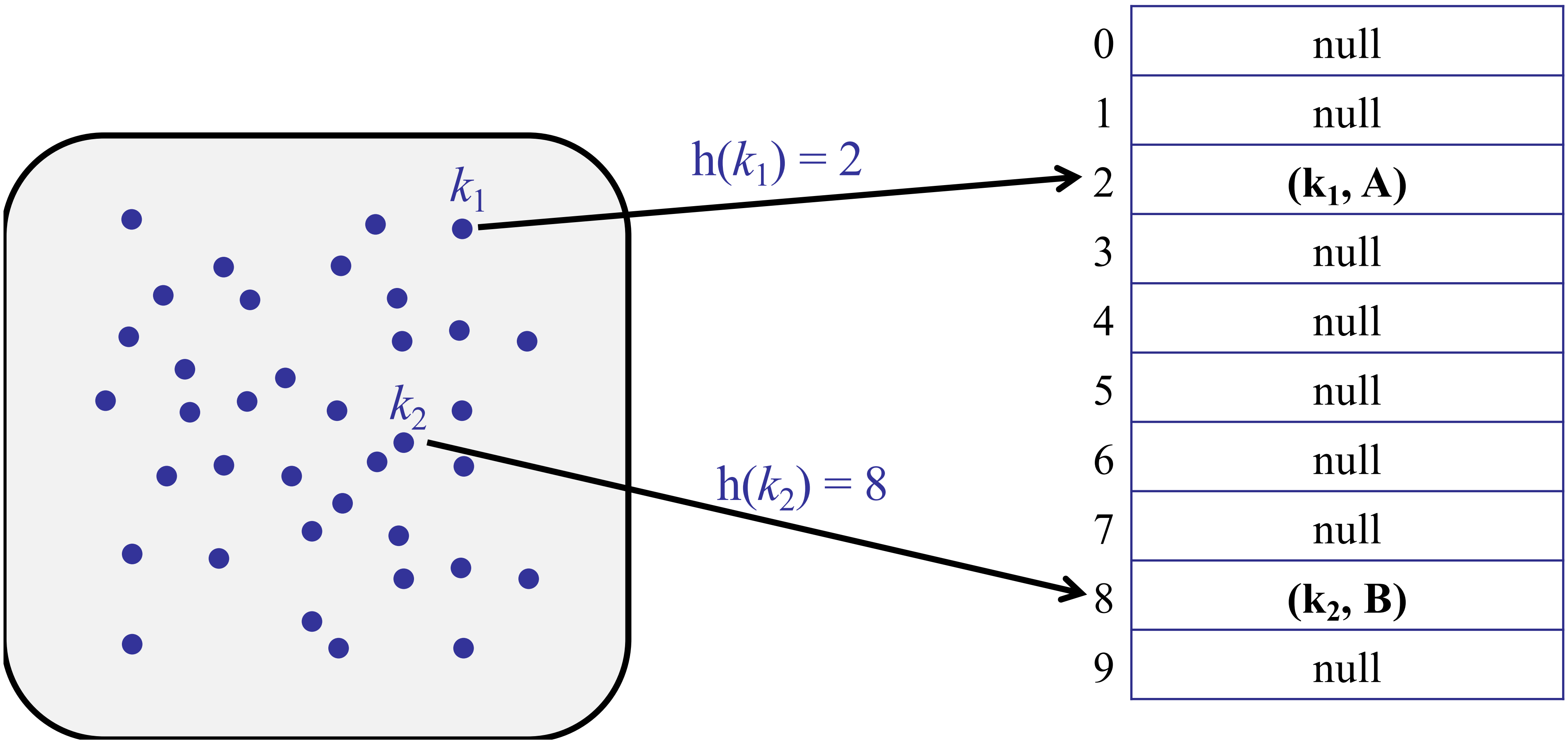
Today

- Java hashing
- Collision resolution: open addressing
- Table (re)sizing

Review

Hash Tables

- Store each item from the symbol table in a table.
- Use hash function to map each key to a bucket.



Resolving Collisions

- Basic problem:
  - What to do when two items hash to the same bucket?
- Solution 1: Chaining
  - Insert item into a linked list.
- Solution 2: Open Addressing
  - Find another free bucket.

Open Addressing

Advantages:

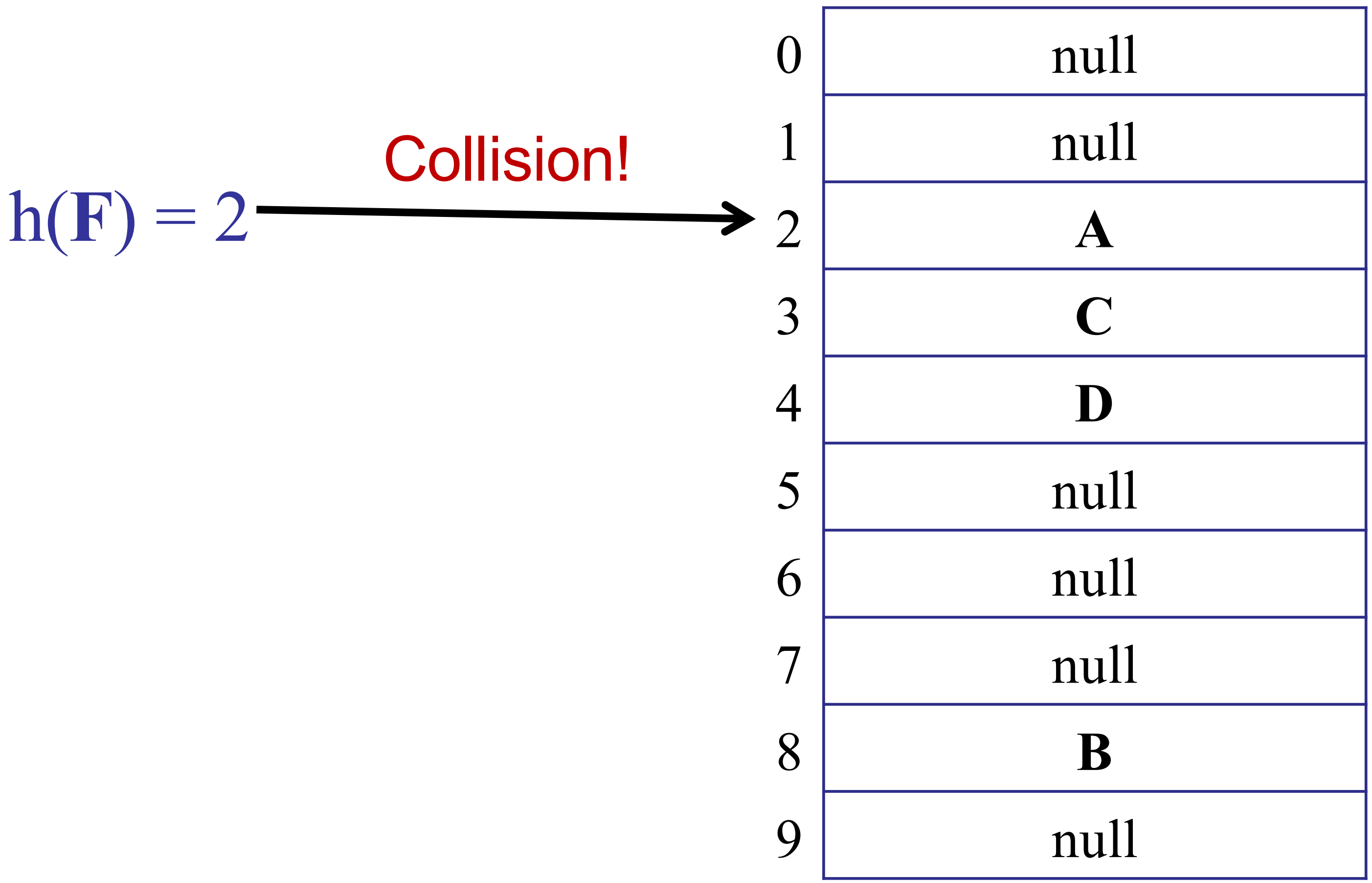
- No linked lists!
- All data directly stored in the table.
- One item per slot.

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

Open Addressing

On collision:

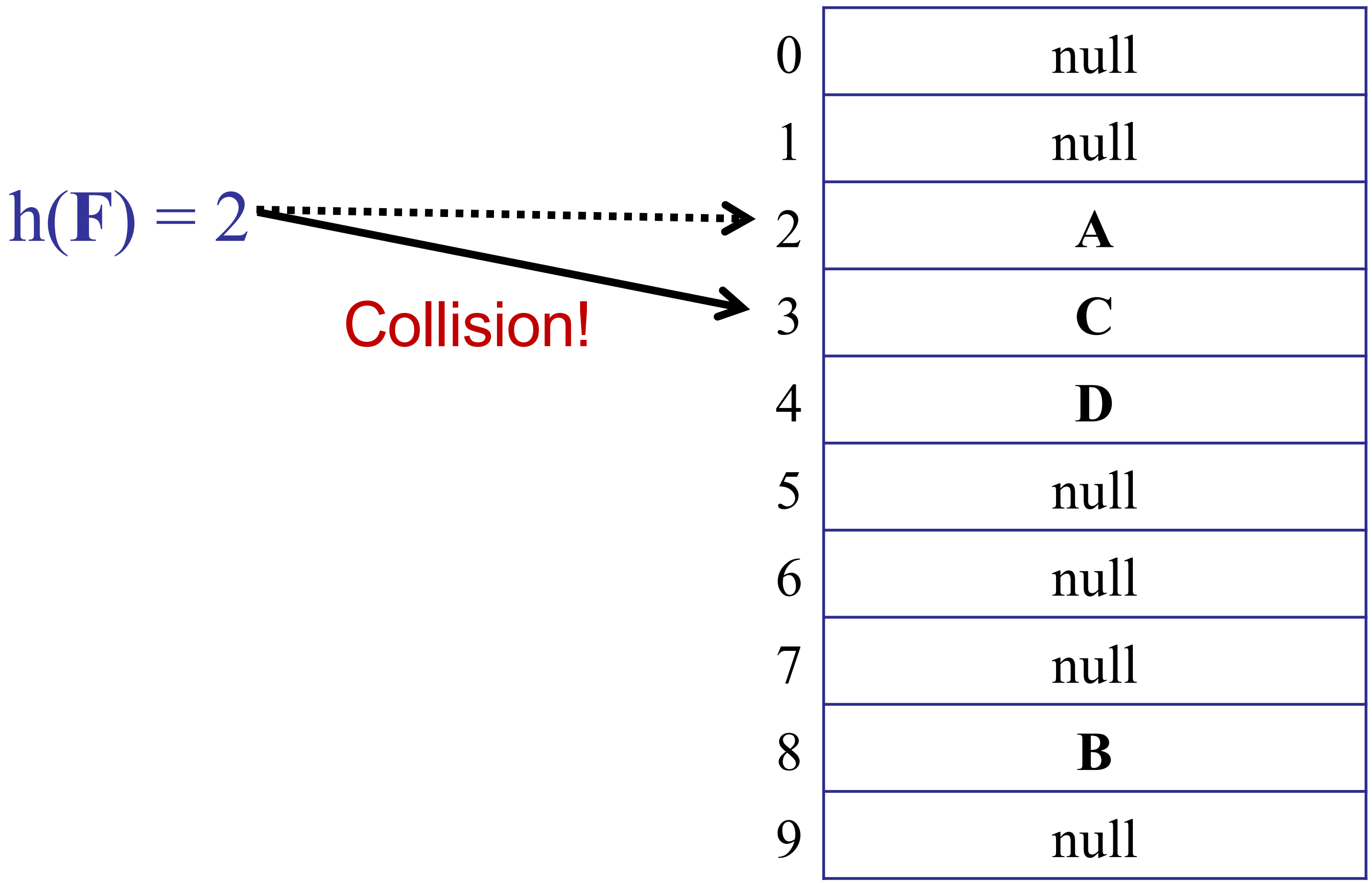
Probe a sequence of buckets until you find an empty one.



Open Addressing

On collision:

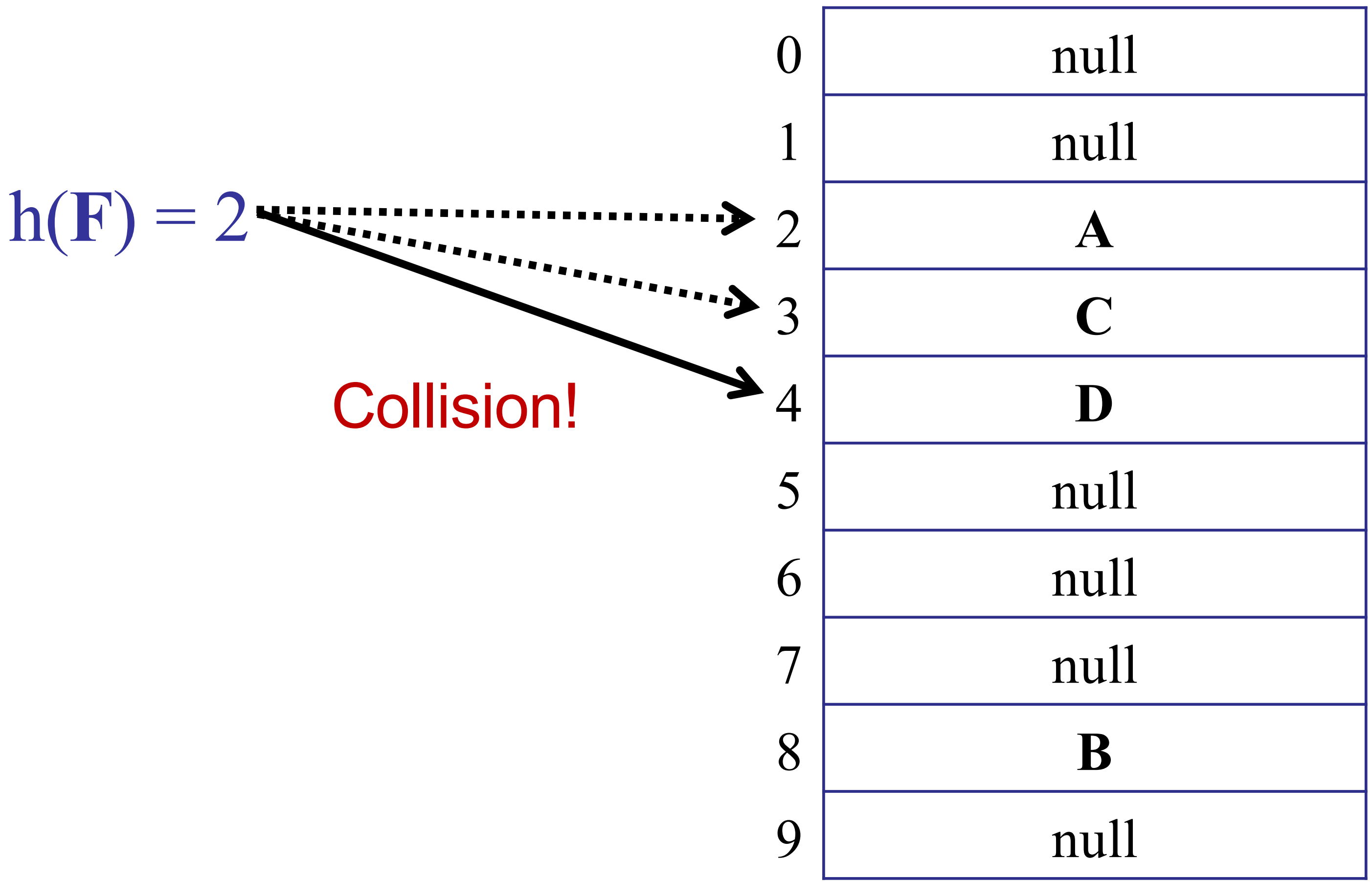
Probe a sequence of buckets until you find an empty one.



Open Addressing

On collision:

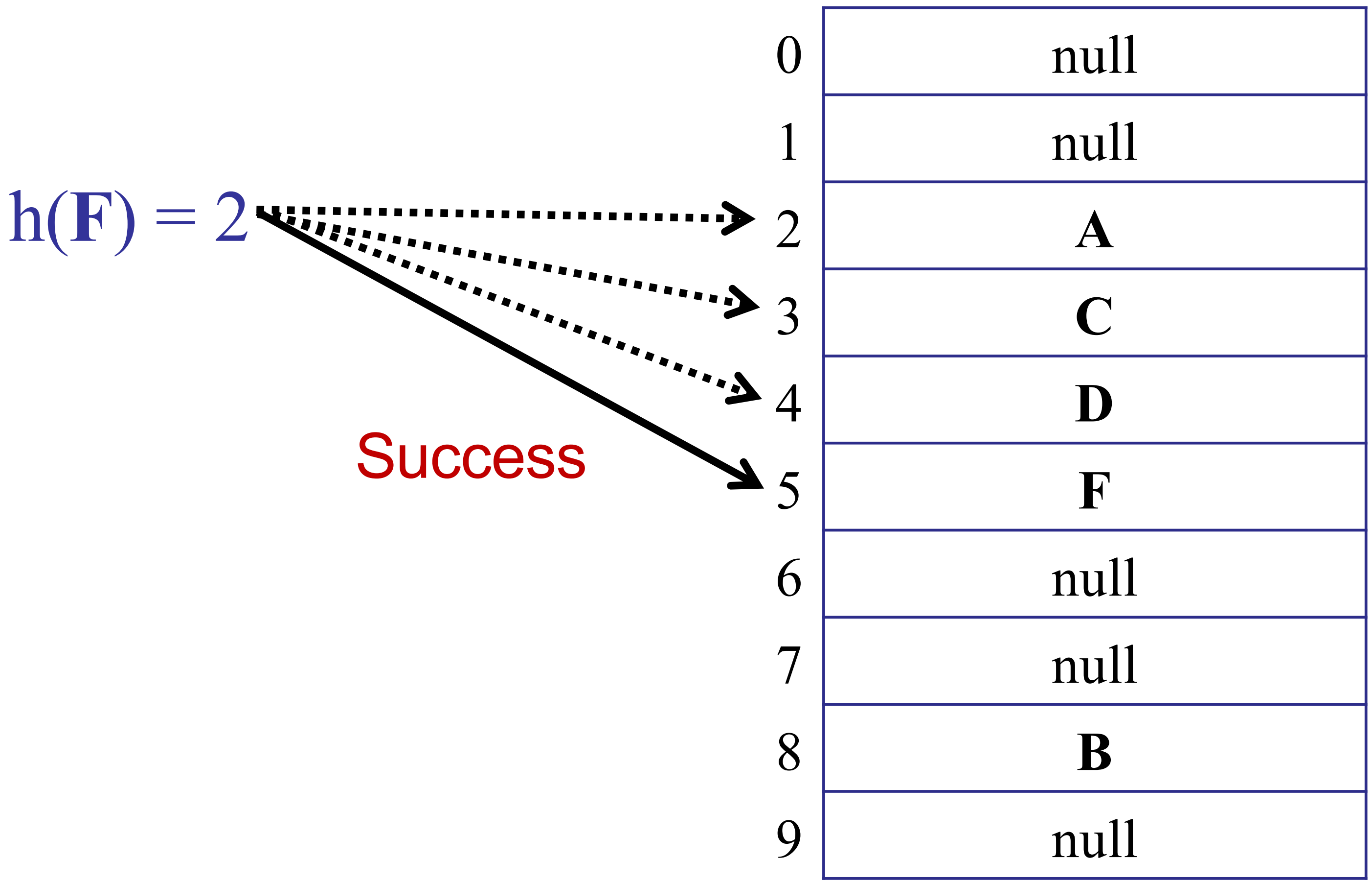
Probe a sequence of buckets until you find an empty one.



Open Addressing

On collision:

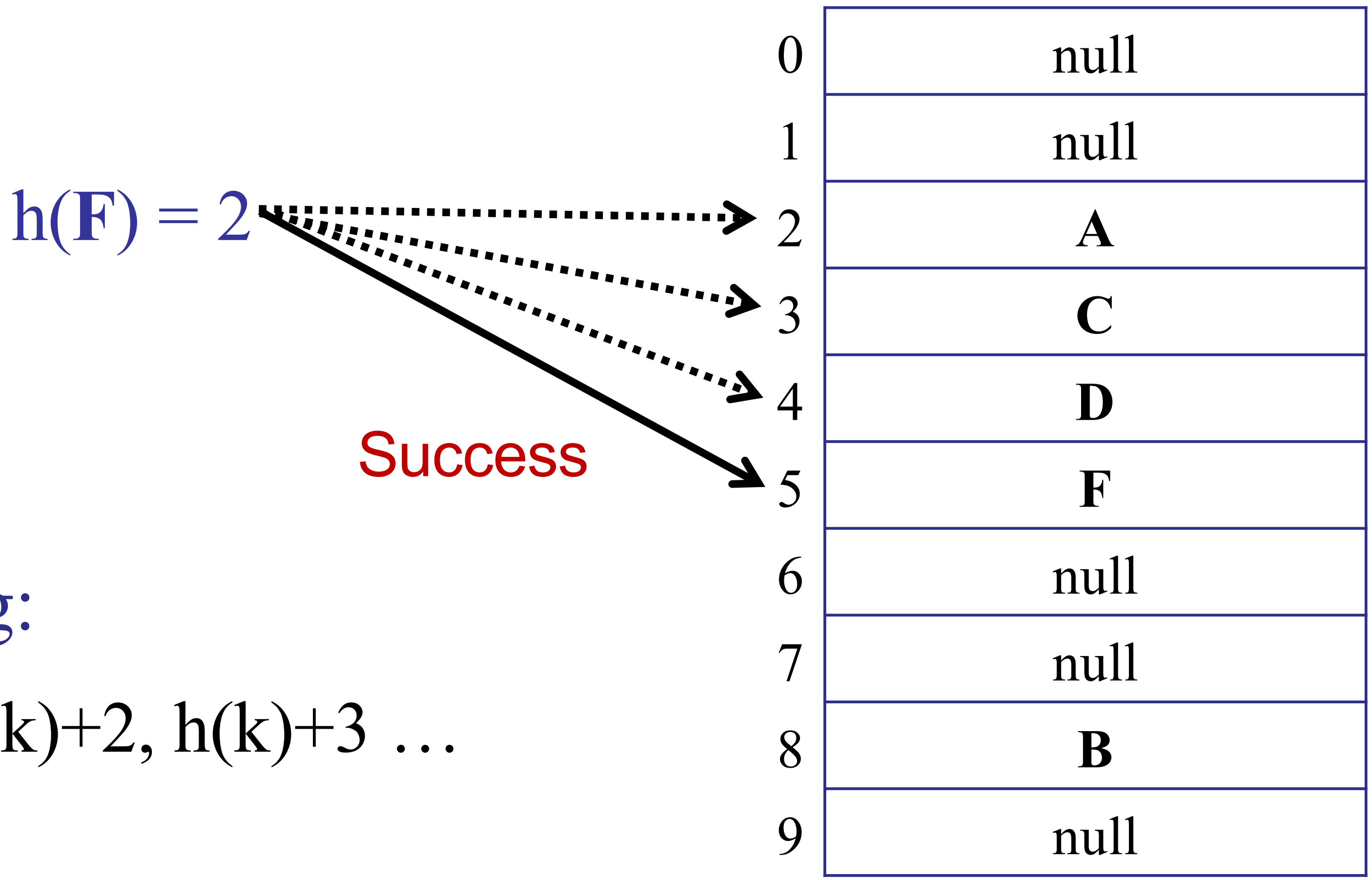
Probe a sequence of buckets until you find an empty one.



Open Addressing

On collision:

Probe a sequence of buckets until you find an empty one.



Linear Probing:

- h(k)+1, h(k)+2, h(k)+3 …

Open Addressing

Hash Function re-defined:

$h(key, i) : U \rightarrow \{1..m\}$

Two parameters:

- key : the thing to map
- i : number of collisions

Open Addressing

Hash Function re-defined:

h(key, i) : U → {1..m}

Example: Linear Probing

- h(k, 1) = hash of key k
- h(k, 2) = h(k, 1) + 1
- h(k, 3) = h(k, 1) + 2
- h(k, 4) = h(k, 1) + 3
- …
- h(k, i) = h(k, 1) + i mod m

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

Open Addressing

Hash Function re-defined:

$h(key, i) : U \rightarrow \{1..m\}$

Example: Weird Probing

- $h(k, 1) = 4$
- $h(k, 2) = 1$
- $h(k, 3) = 8$
- $h(k, 4) = 5$

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

Open Addressing

```
hash-insert(key, data)
int i = 1;
while (i <= m) {                          // Try every bucket
    int bucket = h(key, i);
    if (T[bucket] == null){               // Found an empty bucket
        T[bucket] = {key, data};          // Insert key/data
        return success;                   // Return
    }
    i++;
}
throw new TableFullException();           // Table full!
```

Open Addressing

Hash Function re-defined:

$h(key, i) : U \rightarrow \{1..m\}$

search(key)

- $h(key, 1) = 4$
- $h(key, 2) = 1$
- $h(key, 3) = 8$
- $h(key, 4) = 5$

0	null
1	**G**
2	**A**
3	**C**
4	**D**
5	key
6	null
7	null
8	**B**
9	null

Open Addressing

```
hash-search(key)
int i = 1;
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.
```

Open Addressing

Hash Function re-defined:

$h(key, i) : U \rightarrow \{1..m\}$

search(key)

- $h(key, 1) = 4$
- $h(key, 2) = 1$
- $h(key, 3) = 8$
- $h(key, 4) = 5$

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

Open Addressing

Hash Function re-defined:

$h(key, i) : U \rightarrow \{1..m\}$

delete(key)

- Find key to delete
- Remove it from table.
- Set bucket to null.

0	null
1	**G**
2	**A**
3	**C**
4	**D**
5	**NULL**
6	null
7	null
8	**B**
9	null

What is wrong with delete?

1. Search may fail to find an element.
2. The table will have gaps in it.
3. Space is used inefficiently.
4. If the key is inserted again, it may end up in a different bucket.

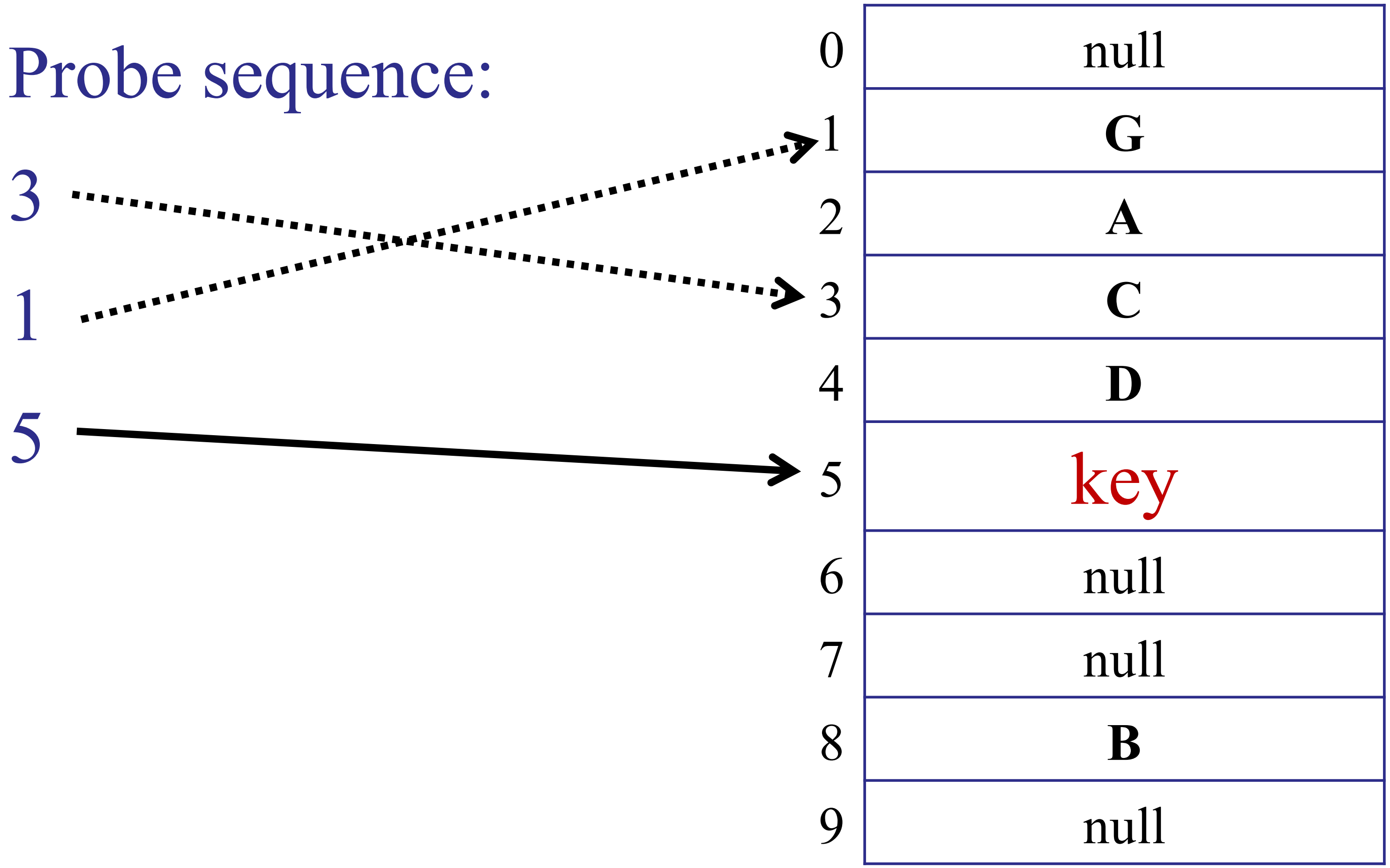
ARCHIPELAGO

is open

Open Addressing

insert(key)

Probe sequence:



Open Addressing

insert(key)

delete(G)

0	null
1	**G → NULL**
2	**A**
3	**C**
4	**D**
5	key
6	null
7	null
8	**B**
9	null

Open Addressing

insert(key)

delete(G)

search(key)

0	null
1	**NULL**
2	**A**
3	**C**
4	**D**
5	key
6	null
7	null
8	**B**
9	null

Open Addressing

insert(key)

delete(G)

search(key)

Probe sequence:

3

1

5

0	null
1	**NULL**
2	**A**
3	**C**
4	**D**
5	key
6	null
7	null
8	**B**
9	null

Open Addressing

insert(key)

delete(G)

search(key)

Probe sequence:

3

1

Not found!

0	null
1	**NULL**
2	**A**
3	**C**
4	**D**
5	key
6	null
7	null
8	**B**
9	null

Open Addressing

Hash Function re-defined:

$$h(key, i) : U \rightarrow \{1..m\}$$

delete(key)

- Find key to delete
- Remove it from table.
- Set bucket to DELETED.

(Tombstone value.)

0	null
1	**G**
2	**A**
3	**C**
4	**D**
5	**DELETED**
6	null
7	null
8	**B**
9	null

Open Addressing

insert(key)

delete(G)

search(key)

Probe sequence:

3

1

5

0	null
1	**DELETED**
2	**A**
3	**C**
4	**D**
5	key
6	null
7	null
8	**B**
9	null

What happens when an insert finds a DELETED cell?

1. Overwrite the deleted cell.
2. Continue probing.
3. Fail.

ARCHIPELAGO

is open

Hash Functions

Two properties of a good hash function:

1. h(*key*, *i*) enumerates all possible buckets.
   - For every bucket *j*, there is some *i* such that:

     $$h(key, i) = j$$

   - The hash function is permutation of $\{1..m\}$.
   - For linear probing: true!

What goes wrong if the sequence is not a permutation?

1. Search incorrectly returns key-not-found.
2. Delete fails.
3. Insert puts a key in the wrong place
4. Returns table-full even when there is still space left.

ARCHIPELAGO

is open

Hash Functions

Two properties of a good hash function:

2. Simple Uniform Hashing Assumption

   Every key is equally likely to be mapped to every bucket, independently of every other key.

For h(*key*, 1)?

For every h(*key*, *i*)?

Hash Functions

Two properties of a good hash function:

2. Uniform Hashing Assumption

   Every key is equally likely to be mapped to every ***permutation***, independent of every other key.

n! permutations for probe sequence: e.g.,

- 1 2 3 4
- 1 2 4 3
- 1 4 2 3
- 1 4 3 2
- …

Hash Functions

Two properties of a good hash function:

2. Uniform Hashing Assumption

   Every key is equally likely to be mapped to every ***permutation***, independent of every other key.

n! permutations for probe sequence: e.g.,

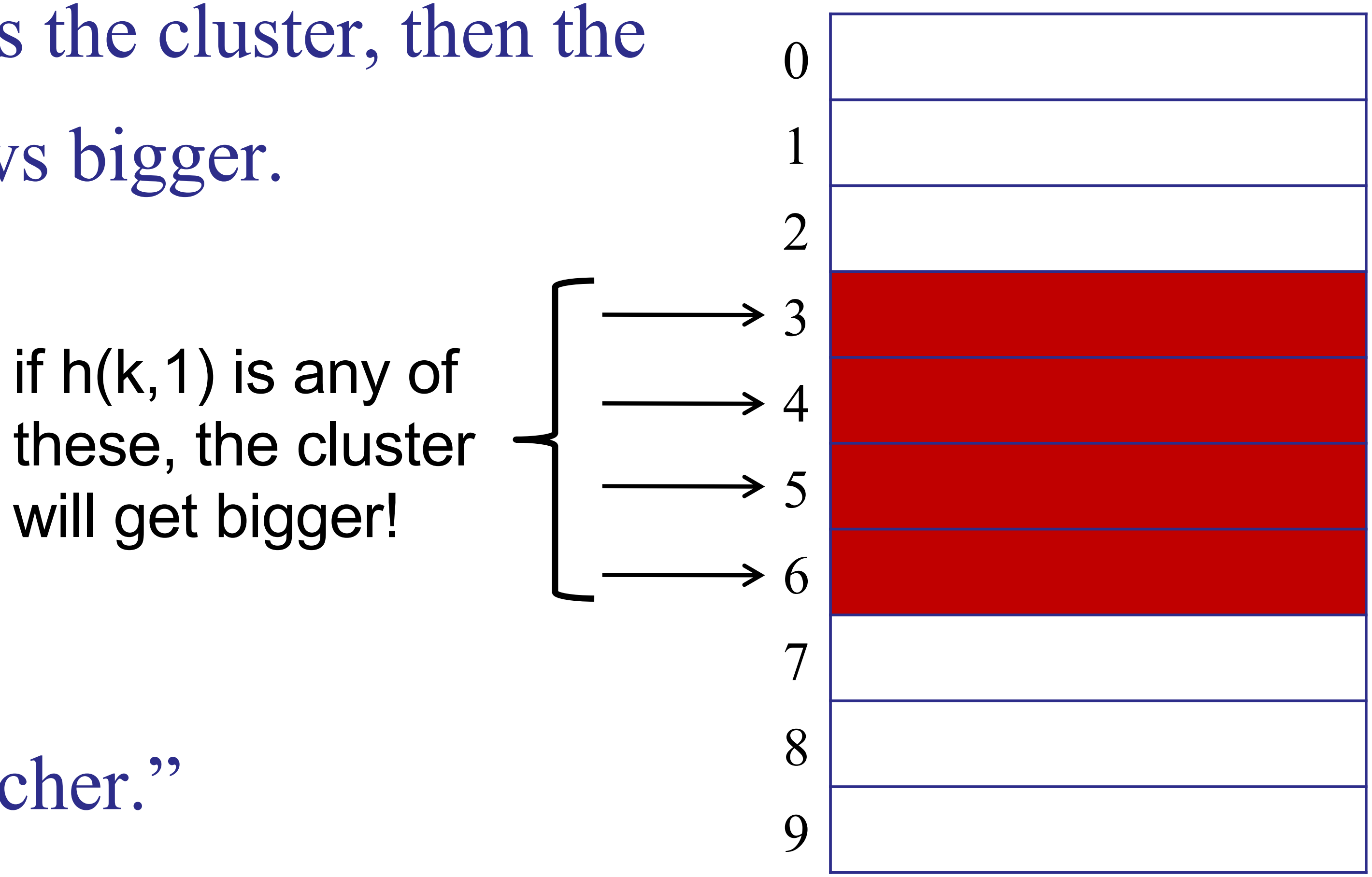
- 1 2 3 4 Pr(1/m)
- 1 2 4 3 Pr(0)
- 1 4 2 3 Pr(0)
- 1 4 3 2 Pr(0)
- …

NOT Linear Probing

Linear Probing

Problem with linear probing: ***clusters***

- If there is a cluster, then there is a higher probability that the next h(k) will hit the cluster.
- If h(k,1) hits the cluster, then the cluster grows bigger.



- "Rich get richer."

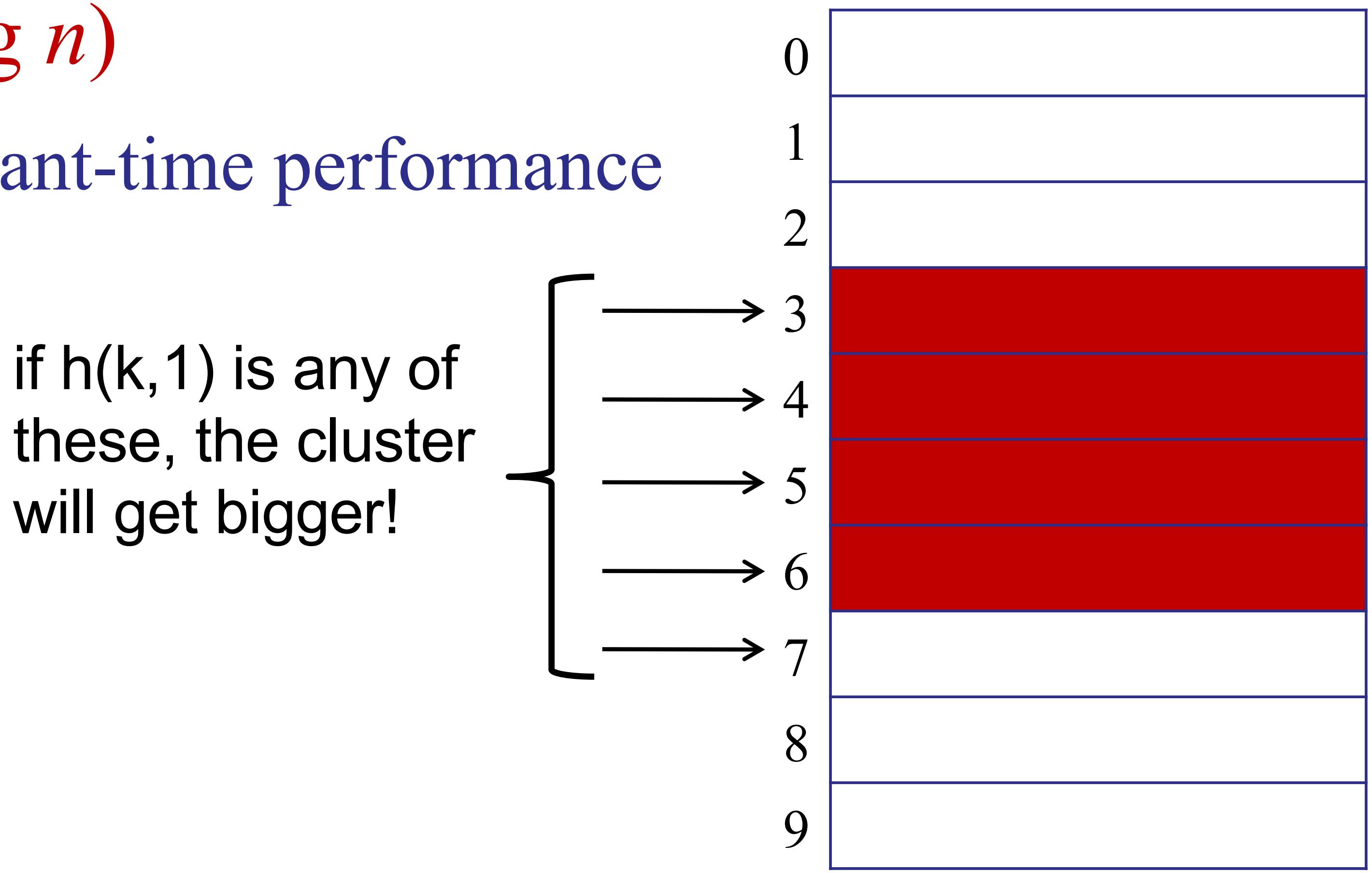
Linear Probing

Problem with linear probing: ***clusters***

- If the table is 1/4 full, then there will be clusters of size:

  $$\theta(\log n)$$

- Ruins constant-time performance



Linear probing

In practice, linear probing is faster!

- Why? Caching!
- It is *cheap* to access nearby array cells.
  - Example: access T[17]
  - Cache loads: T[10..50]
  - Almost 0 cost to access T[18], T[19], T[20], …
- If the table is 1/4 full, then there will be clusters of size: $\theta(\log n)$
  - Cache may hold entire cluster!
  - No worse than wacky probe sequence.

That conversation again…

Professor (for the last 30 years):

"Linear probing is bad because it leads to clusters and bad performance. We need uniform hashing."

Punk in the front row:

"But I ran some experiments and linear probing seems really fast."

Professor:

"Maybe your experiments were too small, or just weren't very well done. Let me prove to you that uniform hashing is good."

Punk in the front row goes and starts a billion dollar startup doing high performance data processing.

Student sitting next to punk in the front row goes to grad school and proves that linear probing really is faster.

Open Addressing

Properties of a good hash function:

2. Uniform Hashing Assumption

   Every key is equally likely to be mapped to every ***permutation***, independent of every other key.

n! permutations for probe sequence: e.g.,

- 1 2 3 4
- 1 2 4 3
- 1 4 2 3
- 1 4 3 2
- …

Double Hashing

- Start with two ordinary hash functions:

$$\mathrm{f}(k),\ \mathrm{g}(k)$$

- Define new hash function:

$$\mathrm{h}(k, i) = \mathrm{f}(k) + i \cdot \mathrm{g}(k) \mod m$$

- Note:
  - Since f(k) is good, f(k, 1) is "almost" random.
  - Since g(k) is good, the probe sequence is "almost" random.

Double Hashing

Hash function

$$h(k, i) = f(k) + i \cdot g(k) \mod m$$

Claim: if g(k) is relatively prime to m, then $h(k, i)$ hits all buckets.

- Assume not: then for some distinct $i, j < m$:

$$f(k) + i \cdot g(k) = f(k) + j \cdot g(k) \mod m$$

➔ $i \cdot g(k) = j \cdot g(k) \mod m$

➔ $(i - j) \cdot g(k) = 0 \mod m$

➔ g(k) not relatively prime to m, since $(i-j \neq 0 \bmod m)$

Double Hashing

Hash function

$$h(k, i) = f(k) + i \cdot g(k) \mod m$$

Claim: if $g(k)$ is relatively prime to m, then $h(k, i)$ hits all buckets.

Example: if $(m = 2^r)$, then choose $g(k)$ odd.

Performance of Open Addressing

If (m==n), what is the expected insert time, under uniform hashing assumption?

1. O(1)
2. O(log n)
3. O(n)
4. O(n^2)
5. None of the above.

ARCHIPELAGO

is open

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.
  - We cannot insert any more items.
  - We cannot search efficiently.

Performance of Open Addressing

Define:

- Load $\alpha = n / m$

Average # items / bucket

- Assume $\alpha < 1$.

Performance of Open Addressing

Define:

- Load $\alpha = n / m$ ← Average # items / bucket
- Assume $\alpha < 1$.

Claim:

For n items, in a table of size m, assuming *uniform hashing*, the expected cost of an operation is:

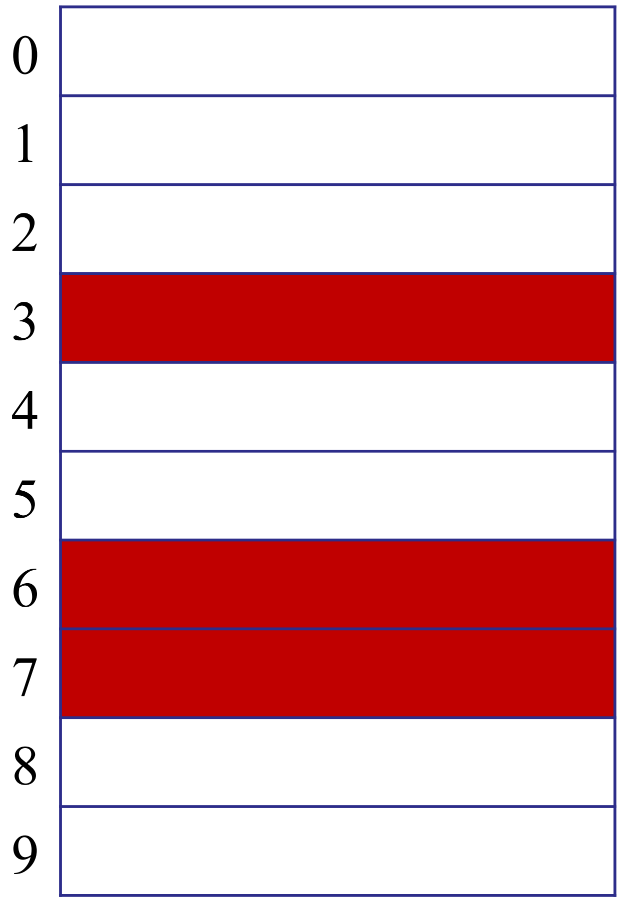
$$\leq \frac{1}{1-\alpha}$$

Example: if (α=90%), then E[# probes] = 10

Performance of Open Addressing

Proof of Claim:

- First probe: probability that first bucket is full is: n/m

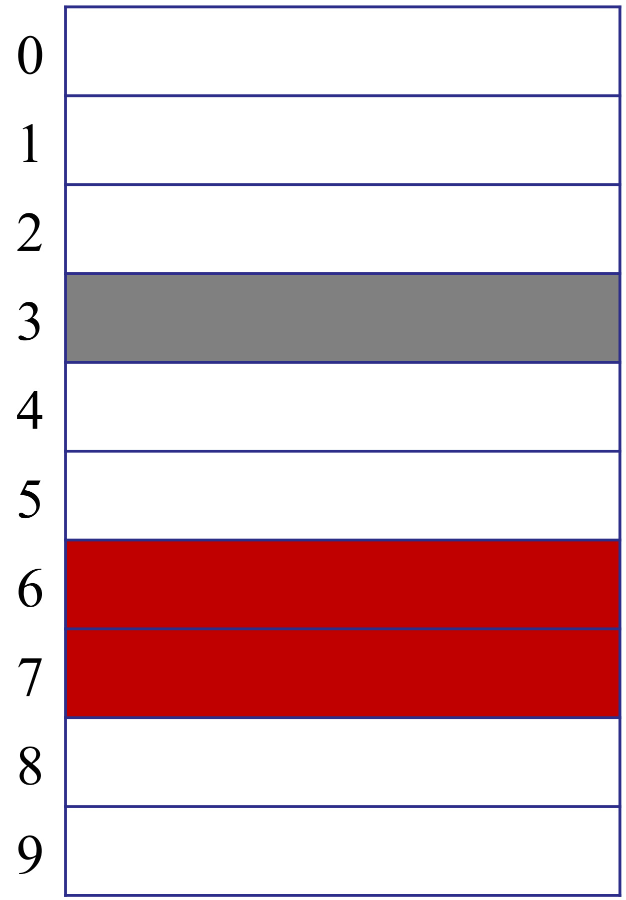


Performance of Open Addressing

Proof of Claim:

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



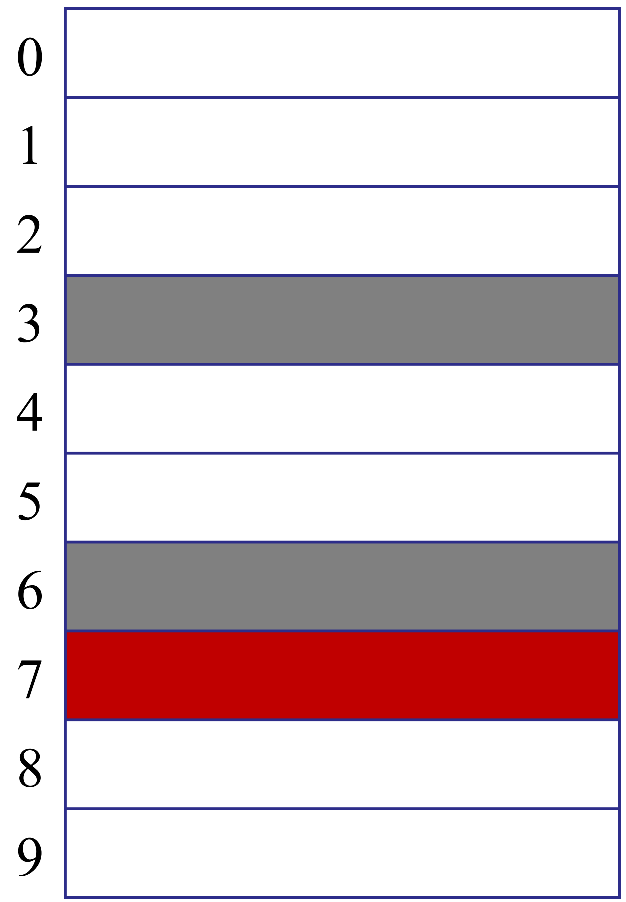
Performance of Open Addressing

Proof of Claim:

– 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

Proof of Claim:

– Expected cost:

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

First probe

Probability of collision on first probe

Performance of Open Addressing

Proof of Claim:

– Expected cost:

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

First probe

Probability of collision on first probe

Probability of collision on second probe

Performance of Open Addressing

Proof of Claim:

- Expected cost:

$$\underbrace{1+\frac{n}{m}}_{\text{First probe}}\left(\underbrace{1+\frac{n-1}{m-1}}_{\text{Second probe}}\left(\underbrace{1+\frac{n-2}{m-2}}_{\text{Third probe}}\left(\quad \cdots\cdots\cdots \quad\right)\right)\right)$$

Performance of Open Addressing

Proof of Claim:

- Expected cost:

$$1+\frac{n}{m}\left(1+\frac{n-1}{m-1}\left(1+\frac{n-2}{m-2}\left(\quad\cdots\cdots\cdots\quad\right)\right)\right)$$

- Note:

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

Performance of Open Addressing

Proof of Claim:

– Expected cost:

$$1+\frac{n}{m}\left(1+\frac{n-1}{m-1}\left(1+\frac{n-2}{m-2}\left(\quad \cdots\cdots\cdots \quad\right)\right)\right)$$

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

Performance of Open Addressing

Proof of Claim:

- Expected cost:

$$1+\frac{n}{m}\left(1+\frac{n-1}{m-1}\left(1+\frac{n-2}{m-2}\left(\quad\cdots\cdots\cdots\quad\right)\right)\right)$$

$$\le 1+\alpha\left(1+\alpha\left(1+\alpha\left(\cdots\cdots\right)\right)\right)$$

$$\le 1+\alpha+\alpha^2+\alpha^3+\cdots$$

Performance of Open Addressing

Proof of Claim:

- Expected cost:

$$1+\frac{n}{m}\left(1+\frac{n-1}{m-1}\left(1+\frac{n-2}{m-2}\left(\quad\cdots\cdots\cdots\quad\right)\right)\right)$$

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

$$\leq 1+\alpha+\alpha^2+\alpha^3+\cdots$$

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

Performance of Open Addressing

Define:

- Load $\alpha = n / m$ ← Average # items / bucket
- 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 (α=90%), then E[# probes] = 10

Advantages…

Open addressing:

- 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.

Disadvantages…

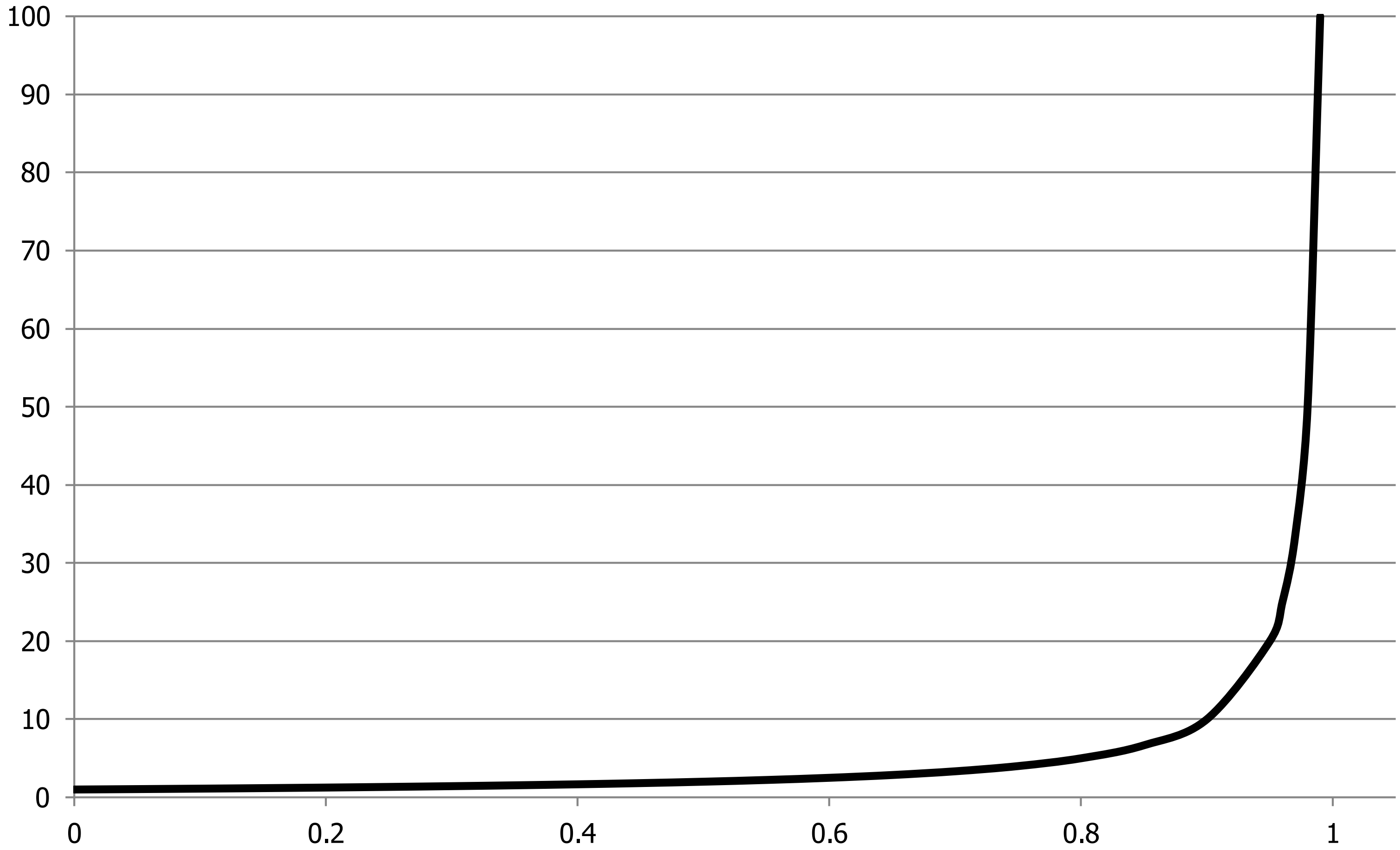
Open addressing:

- 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 $\alpha \rightarrow 1$.

Disadvantages…

Open addressing:

– Performance degrades badly as $\alpha \rightarrow 1$.



Today

- Java hashing
- Collision resolution: open addressing
- Table (re)sizing