CMPT 225: Data Structures & Programming
– Unit 18 –
Hash Tables

Dr. Jack Thomas
Simon Fraser University

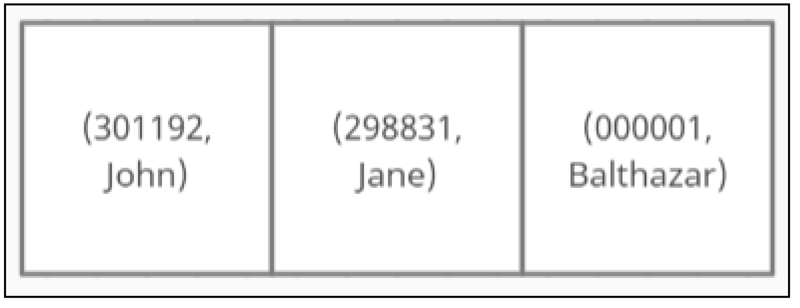
Spring 2021

Today's Topics

- Keys in Maps as Locations
- The Hash Table
- Bucket Arrays and Hash Functions
- Hash Codes
- Compression Functions
- Collision Handling
- Hash Tables in Java

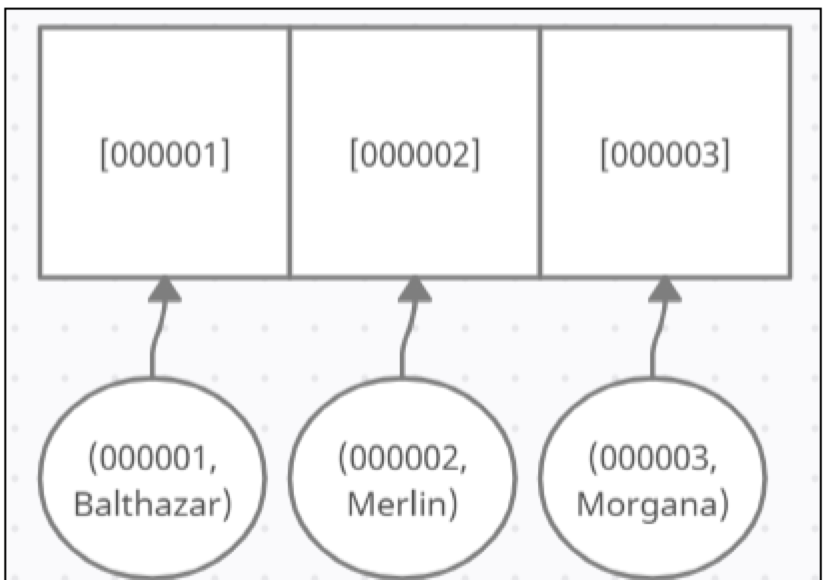
Unique Keys as Addresses

- The big change **Maps** make from previous key-based data structures is that the **keys are unique**.
- The **straightforward way** to think of this is a case where **every key is paired to one piece of data** which now has a unique location in the map – student ID number keys each linking to one student's name, for example.



Unique Keys as Addresses

- **Another way** of thinking about them is each unique key in the map could also mark a location for storing anything that shares that key.

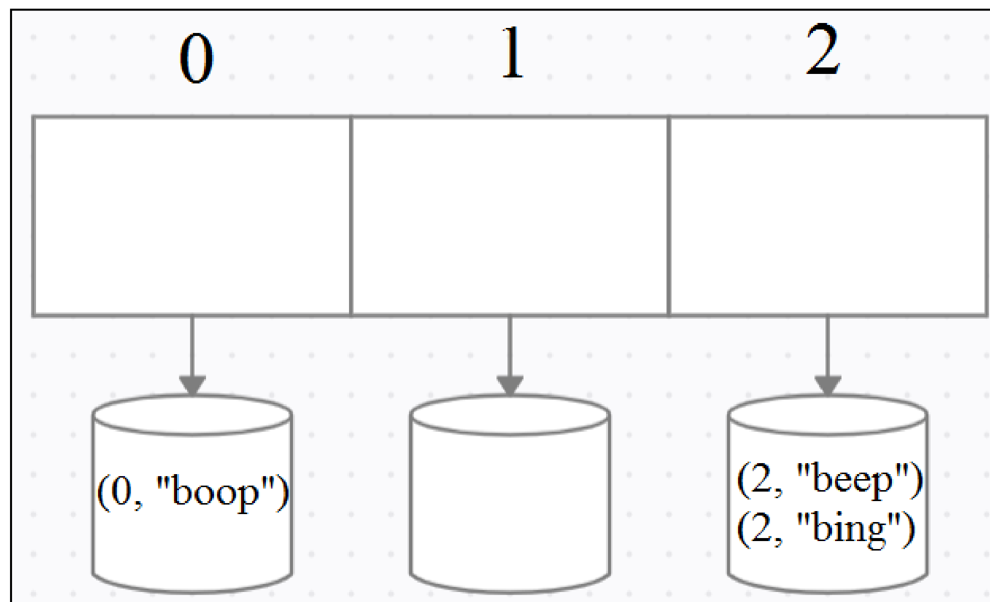


The Hash Table

- A **Hash Table** is a form of Map which **treats each unique key as a pre-existing location**, and stores new entries in these locations based on their key.
- More of **a way to implement a Map** than it is a distinct data structure with its own ADT (sort of like how we handled Heaps).
- Hash Tables are made through combining two components: a **bucket array** and a **hash function**.

Bucket Array

- An **array** A of size N, where each cell of A corresponds with a particular unique key.
- Each cell becomes a “bucket to **store any entry that shares the same key**.
- Ideally, **each bucket should have just one entry in it**, for the fastest and most accurate retrieval.



Drawbacks of the Bucket Approach

- As with regular arrays, **empty cells are wasted space**, which makes the correlation of keys to buckets important for efficiency.
- The number of empty cells will depend on **how the n entries are distributed across the N buckets**, which depends on whatever the chosen keys happen to be.
- What if your data set **doesn't happen to have a well-distributed integer variable** you can just make into a key, anyway?

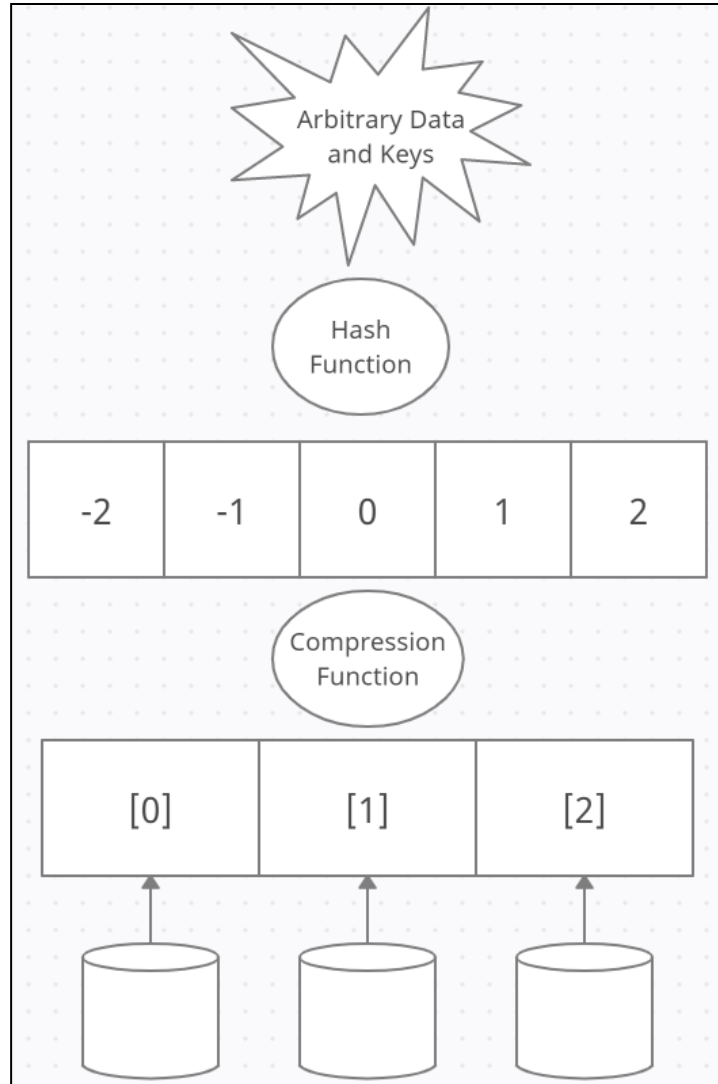
Hash Function

- A **mathematical function** for mapping your chosen key k to an integer from 0 to N-1, **so it can be matched up with a bucket**.
- If we have an **Entry** e with **Value** v and **Key** k that we want to store in our **Hash Table's Bucket Array** A, the **Hash Function** h(k) will give us the index for which bucket in A to store the entity in, so that we can write **e(k,v) -> A[h(k)]**.
- This lets us apply the bucket array to **arbitrary keys**, no matter the kind of data they are (names, dates, several different variables, etc).

Image credit: https://www.grubstreet.com/2015/10/mcdonalds-starts-all-day-breakfast.html

Hash Function

- A hash function has two steps:
  - Mapping the key k to an integer, called the **hash code**.
  - Mapping the hash code to an integer between 0 and N-1 to pair it with a bucket, called the **compression function.**



Hash Codes

- Hash codes have a few distinct properties:
  - They **may be any integer**, even negative, not just the ones from 0 to N-1.
  - The hash function must consistently **produce the same hash code if given the same key** (or a key equal to it, if it’s technically a different instance).
  - Our choice of hash function and key should hopefully give us a unique hash code for every entry, to cut down on **collisions** (two entries with different keys winding up in the same bucket).

Generating Hash Codes in Java

- Java has **a default hashCode() function** built into the Object class which generates a hash code integer based on the memory location for the object.
- **Not always appropriate**, however – two Strings of the same word but stored separately would generate different keys, which is why the String class in Java overrides hashCode() to compare the contents.

Generating Your Own Hash Codes

- **If your key is an integer**, or can be cast to an integer (byte, short, char), you're already done – **that's your hash code**.
- **Floats** have a Float.floatToIntBits(x) function that also gives them a hash-appropriate int representation.
- For **longs** and **doubles**, whose bit representations are twice that of integers, you can sum the integer representation of their first 32 bits with that of their last 32 bits (consider functions like Long.toBinaryString() to get you those bits).

Polynomial Hash Codes

- **Summing components** to produce an integer, like we did with longs and doubles, **won't work if the order of the components matter** (e.g. two String keys, one being "temp01", the other "temp10").
- One hash function for adding multiple components together is a **polynomial hash code**, where each component is multiplied by a constant based on its position in the sequence.
- Intuitively, this **spaces out the integer results depending on their position, reducing the risk of collisions.**

Writing a Polynomial Hash Code

- Say I'm trying to **hash a String**. Each character's integer representation x will be multiplied by some constant, a, to the power of the String's length k minus that character's position in the String:

$$x_0a^{k-1} + x_1a^{k-2} + \ldots + x_{k-2}a + x_{k-1}$$

Polynomial Hash Code Example

- If I were hashing the word "cat" with an a of 33, I could fill out the function like so:

  $x_0a^2 + x_1a + x_2$

  $99*33^2 + 97*33^1 + 116 =$ **111128**

- Studies have shown that, for Strings in English, **some good choices for a** include 33, 37, 39, and 41. These produce relatively few collisions.

What Makes a "Good" Hash Function?

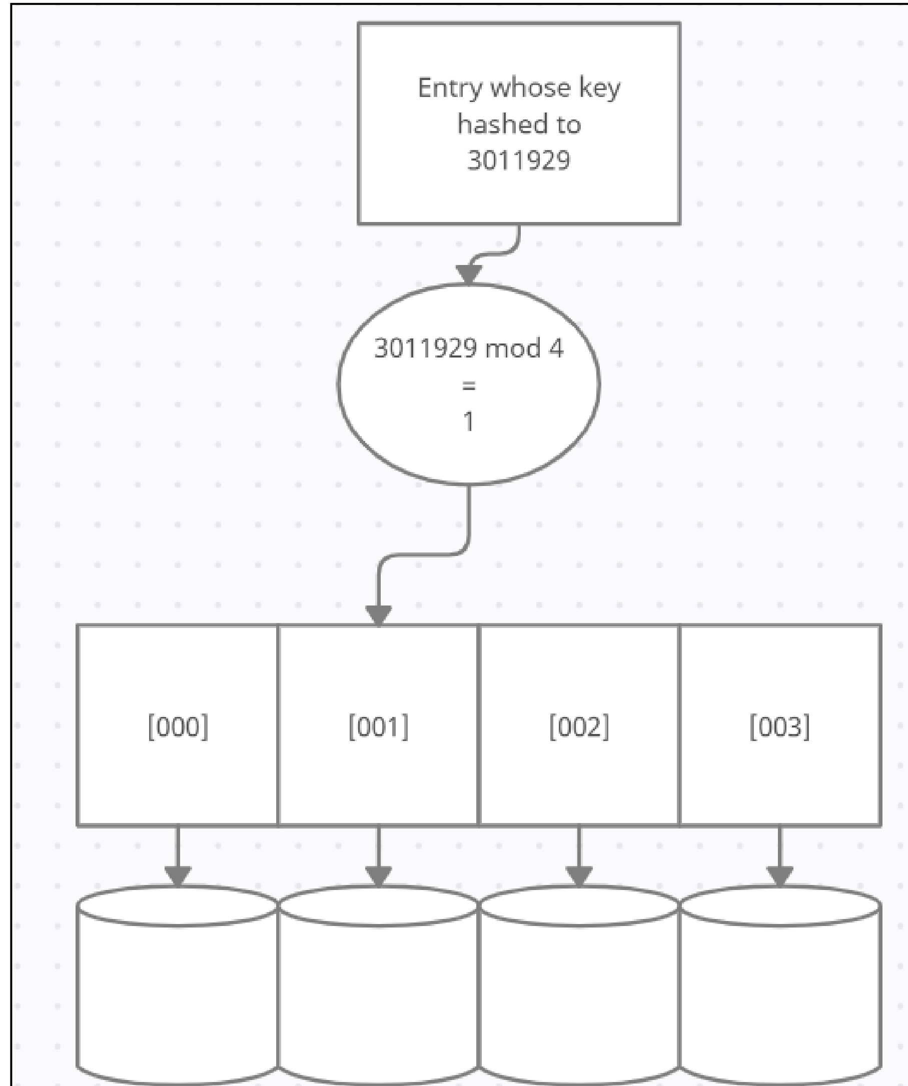
- One that distributes all keys among the available buckets **randomly, but also evenly**.
- To do that, we need a **unique hash code for every key**, and ideally ones that **won't cluster together** into the same bucket after compression.
- We need to **avoid patterns**, which is difficult to solve generally since the keys could be any kind of arbitrary data – using birthdates as a key would mean smoothing out sociological patterns of when people have children!
- Good hash functions for different types of data are **often found experimentally**, with common and important subjects like a particular language having well-known good functions.

Compression Function

- The process of turning the hash code into a bucket for the key's entry.
- As the name implies, this function **compresses the hash code** into a number **between 0 and N-1**.
- This is also where **most collisions occur**, depending on the **ratio between n and N**.
- There are two broad methods used for compression, the **Division Method** and the **MAD Method**.

The Division Method

- The direct method – you've got an integer i hash code and an integer N of buckets, so just evaluate **i mod N**.



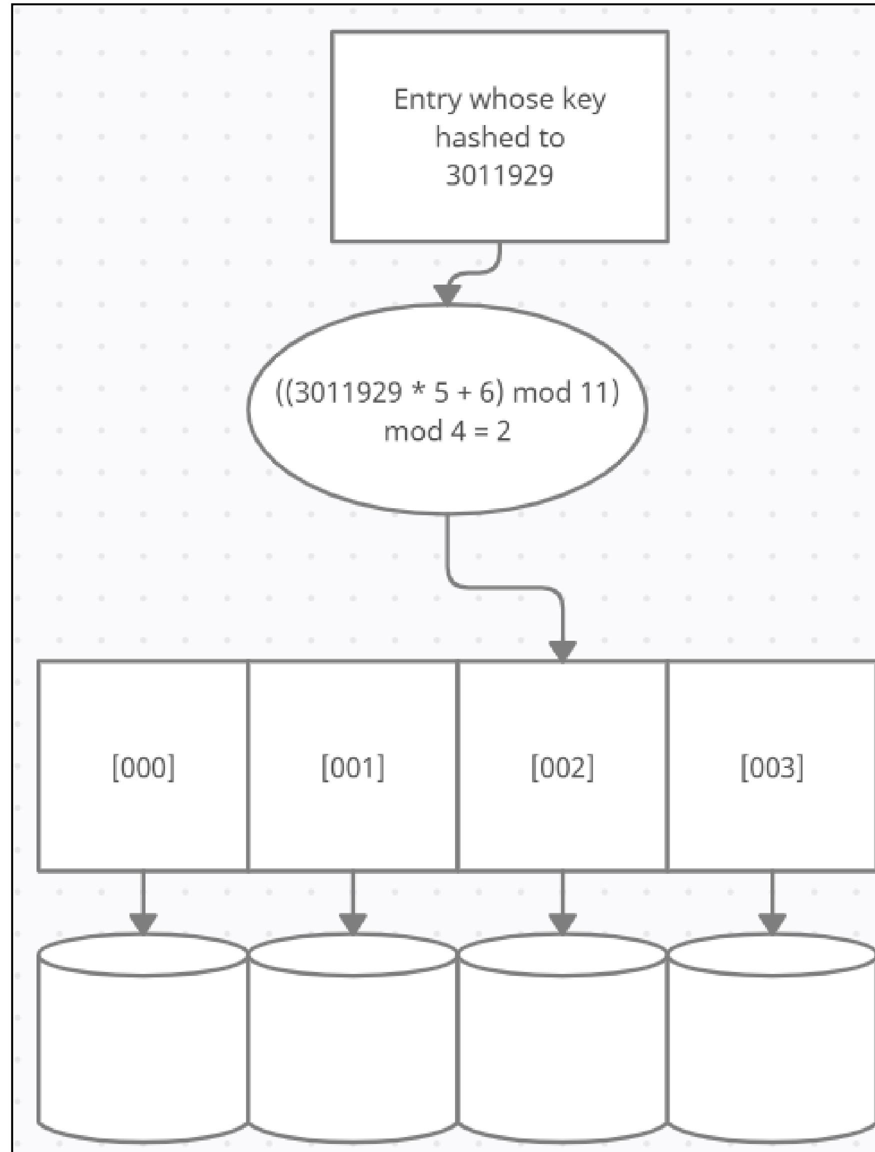
The MAD Method

- A brilliant idea to avoid wars by making it so we’re constantly at risk of Mutually Assured Destruction.

Image credit: https://www.nbcnews.com/id/wbna18237365

The (Actual) MAD Method

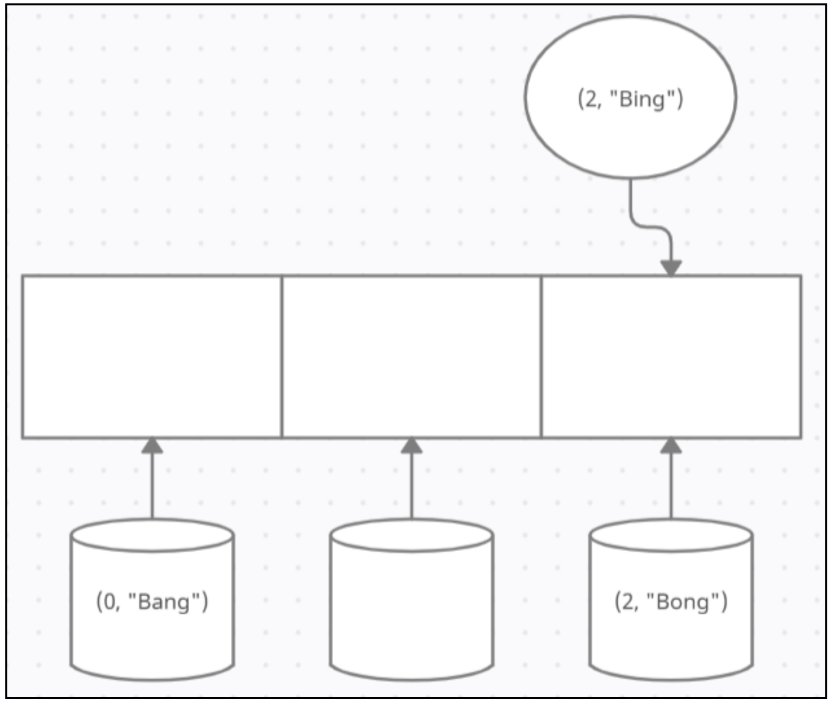
- Stands for **Multiply, Add and Divide**, a more sophisticated method meant to smooth out repeating patterns in a set of keys.
- Given a key integer i, N buckets, a prime number p that's larger than N, and random integers a and b between 0 and p – 1 (with at least a > 0), evaluate **((ai + b) mod p) mod N**.



Collisions

- Maps require keys to be unique, but in the case of Hash Tables, **that only applies to the index of each bucket in the bucket array** – individual entries may have the same key.
- When the hash function produces **the same hash code for two entries**, or the compression function turns **two different hash codes into the same bucket**, this is called a **collision**.

Collisions



- **Collisions are to be avoided where possible**, they hurt the efficiency of the Hash Table, and the choice of hash and compression functions can affect their number.

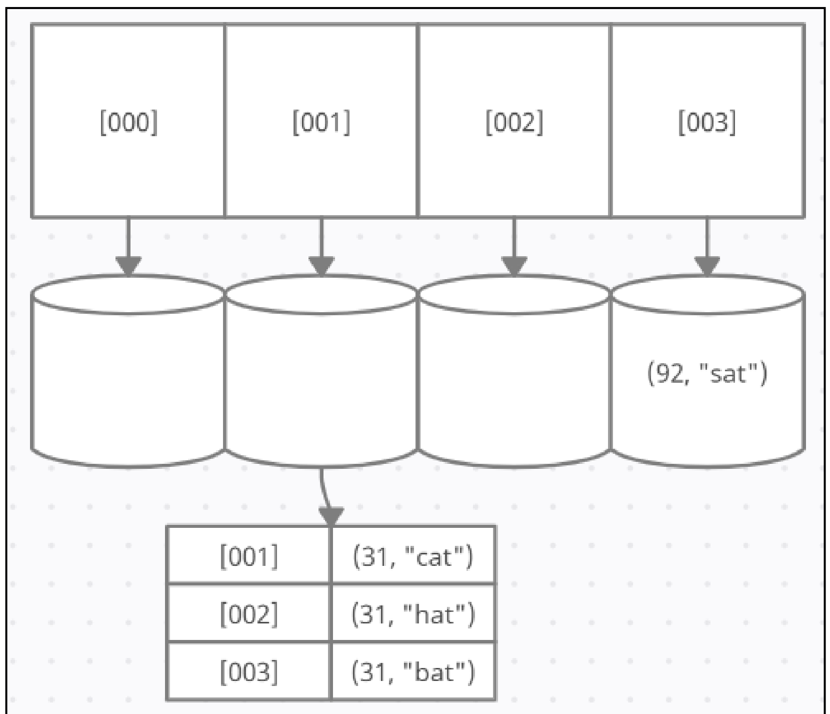
Collision-Handling Schemes

- When collisions do occur, there needs to be a plan for dealing with them.
- It comes down to whether you will accept **multiple entries in the same bucket**, or go **looking for another empty bucket** instead.

1. **Separate Chaining** covers setting up a new data structure within each bucket for storing multiple entries.
2. **Open Addressing** covers a variety of similar techniques for finding a new bucket.

Separate Chaining

- We just accept that a collision has happened and store both (and any subsequent) entries in a Map stored within the bucket.



Maps Within Buckets

- Requires some **extra implementation work**, since now when **adding** to the Hash Table you need to convert the contents of a bucket into a Map if there's already something there.
- When **removing**, you now need to check if a bucket is empty, has one entry, or has a Map of entries, and decide on a rule for how to choose between the entries stored in that Map – randomly? Using a secondary key?

Open Addressing

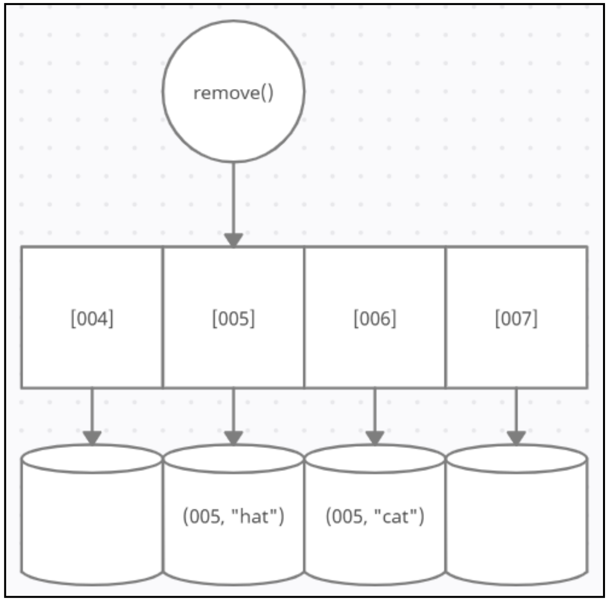
- One drawback to Separate Chaining is it’s space-intensive to have to make a whole Map every time there’s a collision.
- Open Addressing simply tries to find another empty bucket instead.
- It’s not that easy, though – now the way you go looking for a second bucket can also be a source of patterns that cause collisions.

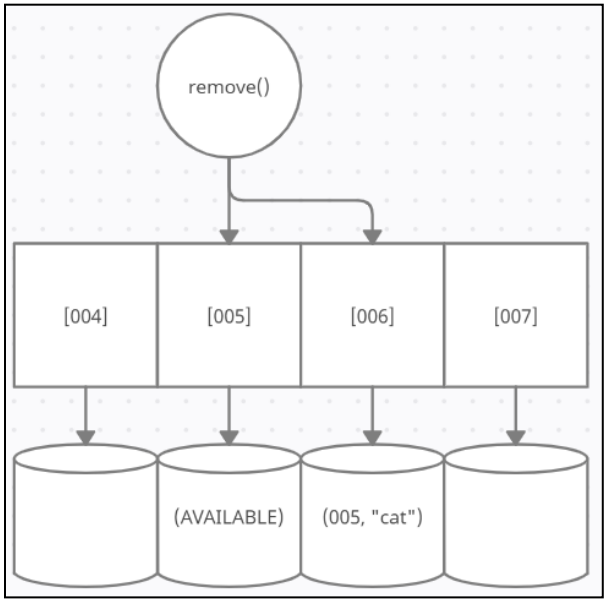
Probing

- Probing strategies start simply checking nearby buckets for the first empty one they can find to put the colliding entry in.
- Linear Probing tries the next bucket after the colliding one, then the next one, and so on.
- Quadratic Probing tries to reduce patterns by checking buckets quadratically (1 away, 4 away, 9 away…).

Probing

- Requires a lot of tweaking to make work, particularly to remove() – now after looking up the original bucket, it may have to retrace the probing pattern looking for the key.
- If it removes an entry, it also has to swap in a special "available" marker instead of a null, so any future remove()'s will keep looking to find its neighbours with the same key.





Double-Hashing

- If your first hash function didn’t find an empty bucket, **try another hash function** instead.
- Add your first hash code to a second one **generated by a second hash function**, then **compress again**.
- If that still doesn’t work, multiply that second hash hash code by 2 and re-add it to the first one and try again, then multiply by 3, and so on.
- Creates many of the **same implementation issues** as Probing (retracing your steps) but may **avoid some clustering patterns**.

The  Factor

- The risk of collisions **grows to an inevitability** as the number of entries n approaches and then passes the number of buckets N.
- This ratio between n and N is called the **Load Factor**, and influences design decisions between the different Hash Table tools we've considered – a low load is space-inefficient, while a high load requires a lot of collision-handling.
- If the Load Factor becomes too high, it's time to increase the number of buckets and completely rebuild the Hash Table – called **Rehashing**.

Hash Tables in Java

- Java does provide a standard Hashtable class, which implements the Map interface (it also extends Dictionary, more on that later!)

```
Hashtable<Integer, String> testHashtable = new Hashtable<Integer, String>();
testHashtable.put(1, "hat");
testHashtable.put(3, "cat");
testHashtable.put(1, "bat");
System.out.println("Contents of Hashtable: " + testHashtable);
```

```
{3=cat, 1=bat}
```

Let’s Implement a Hash Table!

- We’ll write a Hash Table class in Java that implements Map (and therefore also Entry), uses Linear Probing for collisions, MAD for compression, and maintains a Load Factor of 0.5 or less.

```java
public class HashTableMap<K,V> implements Map<K,V> {
    //A blank entry for when we fill in gaps after Linear Probing
    protected Entry<K,V> AVAILABLE = new HashEntry<K,V>( k: null,  v: null);
    //number of entries
    protected int n = 0;
    //The prime factor and capacity of the bucket array
    protected int prime, capacity;
    //The bucket array
    protected Entry<K,V>[] bucket;
    //Shift and scale factors
    protected long scale, shift;
```

```java
//Our entry, set to take any generic key or value we might assign
public static class HashEntry<K,V> implements Map.Entry<K,V> {
    protected K key;
    protected V value;
    public HashEntry(K k, V v) { key = k; value = v;}
    public V getValue() { return value;}
    public K getKey() { return key;}
    public V setValue(V val) {
        V oldValue = value;
        value = val;
        return oldValue;
    }
    /*
    Our comparison function tries to cast whatever it's given to a HashEntry before it
    goes fishing for a key and value, to make sure that whatever it was given can be
    a valid HashEntry and throwing an exception if not.
     */
    public boolean equals(Object o) {
        HashEntry<K,V> ent;
        try { ent = (HashEntry<K,V>) o;}
        catch (ClassCastException ex) { return false;}
        return (ent.getKey() == key) && (ent.getValue() == value);
    }
}
```

```java
//Main constructor, which sets the starting capacity,
//and the prime, then uses the prime to randomly roll
//up the scale and shift factors for hashing.
public HashTableMap(int p, int cap){
    prime = p;
    capacity = cap;
    bucket = (Entry<K,V>[]) new Entry[capacity];
    java.util.Random rand = new java.util.Random();
    scale = rand.nextInt( bound: prime - 1) + 1;
    shift = rand.nextInt(prime);
}
//A version of the constructor with a default prime.
public HashTableMap(int cap) { this( p: 109345121, cap);}
```

```java
//This will come in handy as a helper later
protected void checkKey(K k) throws InvalidKeyException {
    if(k == null) throw new InvalidKeyException("Invalid key: null.");
}
//This generates the bucket for a given key
//We use the default hashCode() function,
//then apply the MAD method compression function
public int hashValue(K key) {
    return (int) ((Math.abs(key.hashCode()*scale + shift) % prime) % capacity);
}
//Standard size and isEmpty functions
public int size() { return n;}
public boolean isEmpty() { return (n == 0);}
```

```java
//For finding an entry based on a key, helps other functions
protected int findEntry(K key) throws InvalidKeyException{
```

1

```java
    int avail = -1; //used as part of Linear Probing
    checkKey(key); //Make sure the key is valid, throw if not
    int i = hashValue(key); //find the bucket based on the key
    int j = i; //We'll need to remember where we started
```

```java
do {
```

2

```java
    //Get the entry at the key's bucket
    Entry<K,V> e = bucket[i];
    //A null entry usually means there's no
    //entry with that key, so we can just
    //return the bucket where it would go
    //and call it a day.
    if (e == null) {
        if (avail < 0)
        {
            //We're going to return the
            //negative of the bucket at
            //the end of the loop to indicate
            //the bucket is empty.
            avail = i;
        }
        break;
    }
```

3

```java
//If there's an entry and the keys
//match, you should also return the
//current bucket
if (key.equals(e.getKey()))
{
    return i;
}
```

4

```java
//If there's an entry but it's our
//special AVAILABLE entry, linearly
//probe the next bucket.
if (e == AVAILABLE)
{
    if (avail < 0)
    {
        avail = i;
    }
}
```

5

```java
//Try the next bucket along, which may require
//wrapping around to the start of the array.
i = (i + 1) % capacity;
```

6

```java
    //Also make sure you haven't checked every bucket.
} while (i != j);
```

```java
public V put (K key, V value) {
    int i = 0;
    //First check if there's already an
    //entry with this key in the hash table.
    try {
        i = findEntry(key);
    } catch (InvalidKeyException e) {
    }
    //This key already exists, which means
    //we're swapping in the new value into
    //the entry and then we're done
    if (i >= 0)
    {
        return ((HashEntry<K,V>) bucket[i]).setValue(value);
    }
```

```java
//The ideal load is 1/2 the capacity or less,
//so we rehash if our new entry pushes the load
// up to 1/2
if (n >= capacity/2)
{
    try {
        rehash();
    } catch (InvalidKeyException e) {
    }
    //You'll need to update the bucket we're
    //going to use if we've just gone and
    //rehashed everything
    try {
        i = findEntry(key);
    } catch (InvalidKeyException e) {
    }
}
```

```java
    //un-flip the bucket's index, since we flipped
    //it earlier to indicate it was empty
    bucket[-i-1] = new HashEntry<K,V>(key, value);
    n++;
    //We didn't swap an old value out if we got this
    //far, so return null.
    return null;
}
```

```java
//For rebuilding the whole hash table twice as big
protected void rehash() throws InvalidKeyException {
    //Double the old capacity, save the old bucket
    //array, and set the hash table to the new,
    //empty, double-sized one.
    capacity = 2*capacity;
    Entry<K,V>[] old = bucket;
    bucket = (Entry<K,V>[]) new Entry[capacity];
    //Reset the scale and shift factors
    java.util.Random rand = new java.util.Random();
    scale = rand.nextInt( bound: prime - 1) + 1;
    shift = rand.nextInt(prime);
    //Go through the old bucket array and find all the
    //real entries, and put them in the new bucket.
    for (int i = 0; i < old.length; i++)
    {
        Entry<K,V> e = old[i];
        if((e != null) && (e != AVAILABLE))
        {
            int j = -1 - findEntry(e.getKey());
            bucket[j] = e;
        }
    }
}
```

```java
//Returns a value based on a key
public V get(Object key) {
    int i = 0;
    try {
        //findEntry() does the
        //heavy lifting here
        i = findEntry((K)key);
    } catch (InvalidKeyException e) { }
    if(i < 0) return null;
    return bucket[i].getValue();
}
```

```java
//Removes an entry based on a key,
//returns the value that was there
public V remove (Object key) {
    int i = 0;
    try {
        i = findEntry((K)key);
    } catch (InvalidKeyException e) {
    }
    if (i < 0 ) return null;
    //Save the value from the entry
    //you're removing
    V toReturn = bucket[i].getValue();
    //Mark it as newly available
    bucket[i] = AVAILABLE;
    n--;
    return toReturn;
}
```

```java
@Override
public boolean containsKey(Object key) {
    return false;
}
@Override
public boolean containsValue(Object value) {
    return false;
}
@Override
public Collection<V> values() {
    return null;
}
@Override
public Set<K> keySet(){
    return null;
}
@Override
public Set<Entry<K, V>> entrySet() {
    return null;
}
@Override
public void putAll(Map<? extends K, ? extends V> m) {
}
@Override
public void clear() {
}
```

```java
HashTableMap<Integer, String> testMap = new HashTableMap<Integer, String>( p: 33,  cap: 10);
testMap.put(9, "First entered");
testMap.put(1, "Second entered");
testMap.put(5, "Third entered");
for (int i = 0; i < 10; i++)
{
    String value = testMap.get(i);
    System.out.println(value);
}
System.out.println();

HashTableMap<String, String> exampleMap = new HashTableMap<String, String>( p: 33, cap: 10);
exampleMap.put("keyword", "first value");
exampleMap.put("anotherKeyword", "second value");
String value = exampleMap.get("keyword");
System.out.println(value);
value = exampleMap.get("anotherKeyword");
System.out.println(value);
```

```
null
Second entered
null
null
null
Third entered
null
null
null
First entered

first value
second value
```

Why Are Hash Tables Efficient?

- The process of converting any key to an assigned bucket, while complicated, is still a **constant time operation**.
- This recreates many **conditions and benefits of an array**, like constant-time accessing, as well as drawbacks like unused space and periodic rebuilding – but **only if collisions remain low**.

Analyzing Hash Tables

- A traditional asymptotic analysis would assume a **worst-case hash function which always collides**, requiring O(n) for every operation.
- Since the entire advantage of Hash Tables is found in good hash functions that minimize collisions, we have to use **average-case analysis** and a bunch of statistical methods that are beyond the scope of this course.
- Studies suggest a load factor of >0.5 for open addressing and >0.9 for separate chaining is optimal. The HashTable class in Java uses 0.75, but it may also be re-specified.

When To Use a Hash Table?

- Ironically, useful for **counting coll**isions, like the frequency of different words in a document.
- **Caches and other quick-access memory solutions** often use hash tables to store and retrieve chunks of data.
- Essentially, **most situations where an array would be useful**, except instead of tracking entries using their position in the array, you can use a key.

Recap: Hashing it Out

- **Hash Tables** are an implementation of **Maps** that treats each unique key as a location.
- It combines **a Bucket Array** and **Hash Function** to match any arbitrary key with a (hopefully unique) location in the data structure.
- Hash functions first generate a **Hash Code** from the arbitrary key, then run it through the **Compression Function** to match it to a bucket.
- **Collisions** occur when two different entries get assigned the same bucket, and require a **Collision Handling Scheme** to manage.
- Good hash functions will **minimize the number of collisions**.
- There's a lot of design and implementation questions around making a Hash Table, but thankfully **Java has a standard one**.