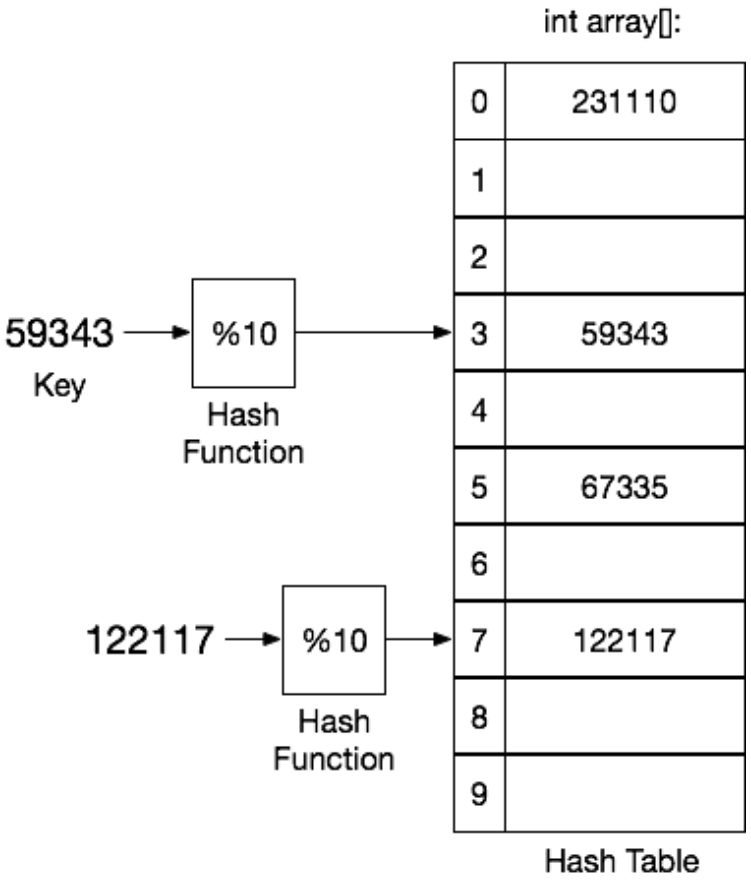


11.
Hash Table

ECE 309

Dr. Greg Byrd

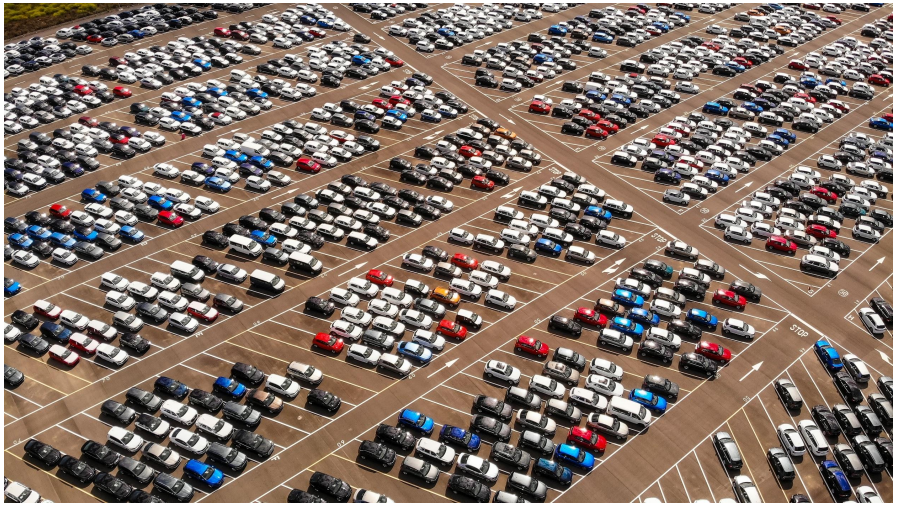
Hash Table overview

A ***hash table*** is a data structure that stores unordered items by mapping (or hashing) each item to a location in an array.

The key advantage is that **searching** for an item is **O(1)**.

Motivating Example: Car Dealership

Where's the key for car #197807697?



Let's assume there are 10,000 cars on any given day on the car lot. Salesperson finds car on computer, needs key for a test drive.

Each car has a 9-digit ID.

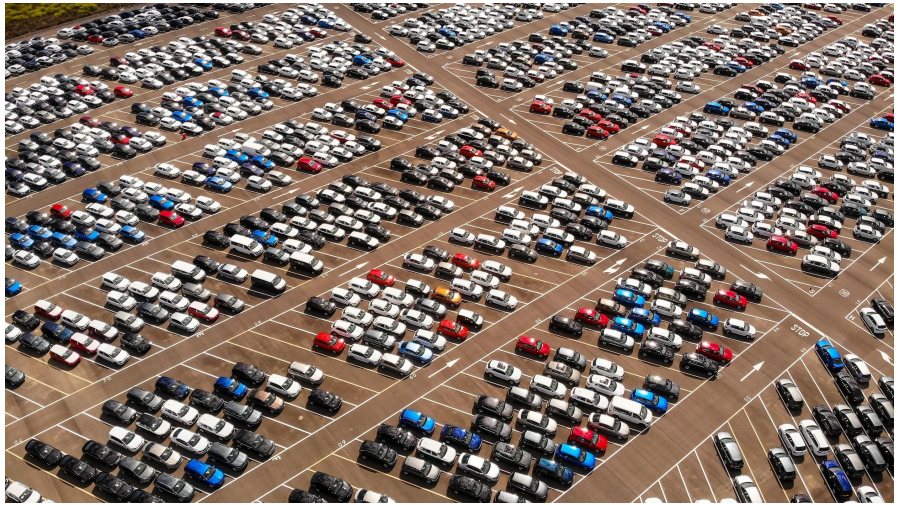
Photo by Carles Rabada on Unsplash

Option #1: Box of keys

Where's the key for car #197807697?

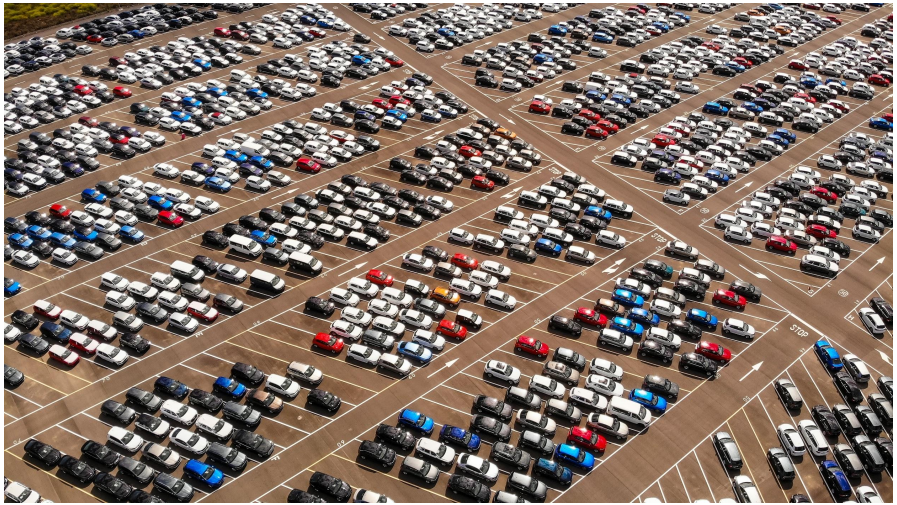
All the keys are in one box.

How long will it take to find the key?
(Best case? Avg case? Worst case?)



Option #2: Box of keys per color

Where's the key for car #197807697?



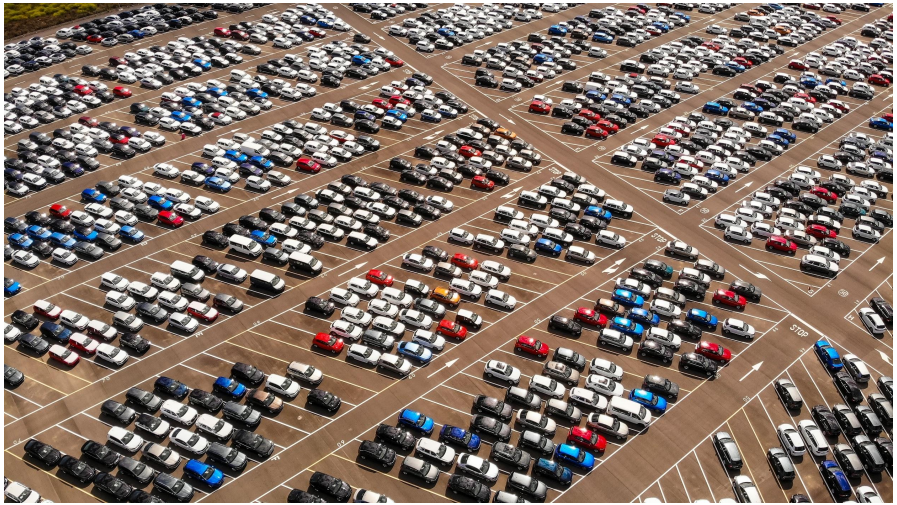
One box for each color. How many boxes? How do you know color?

How long will it take to find the key?

(Best case? Avg case? Worst case?)

Option #3: 1000 boxes of keys

Where's the key for car #197807697?



Use the last three digits (id % 1000) to choose a box.

How long will it take to find the key?

(Best case? Avg case? Worst case?)

Hash Table

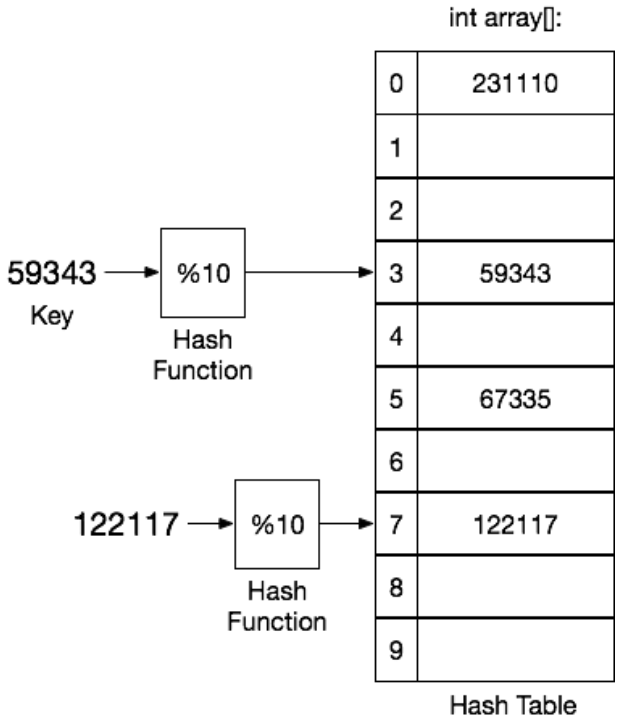
Divide search space into a number of smaller spaces, called buckets.

Hash function maps a key (id) to a particular bucket.

Searching a bucket is much less expensive than searching the entire space.

Photo by Jessica Johnston on Unsplash

Parts of a hash table



Hash Table holds {231110, 59343, 67335, 122117}

Data is what we want to put in the table:

- It can be anything.
- It's often a pointer to avoid copying objects.
- The example uses an int as the data.

The key is used to identify each data value:

- The key should be **unique** for each unique data item.
- Key and data can be the same, as shown to the left.
- Otherwise, data type should provide a function for extracting the key from the data.

Hash Function: turns the key into an index in the hash table array.

Hash Table is an array that holds the data.

- Each entry in the table is called a **bucket**

Common operations on a hash table

insert(key,data): add data to the hash table using key

- If key and data are the same, only one argument is needed
- Optional: whether to allow multiple insertions of the same data or not

search(key): check to see if data with corresponding key is in the hash table

- Many options for the return value of search: the data, true/false, an iterator

remove(key): remove all data with corresponding key

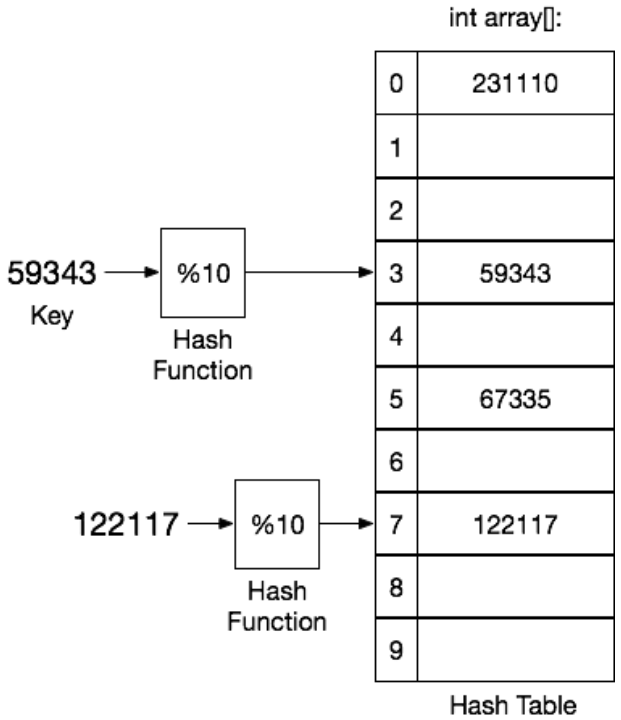
Example of operations

Consider a hash table holding integers. Think of it as a set.

Key is equal to the data. Might or might not allow duplicate values.

Operation	State of Hash Table	Result
insert(10)	{5, 19}	return true; {5, 19, 10}
insert(10)	{5, 10, 19}	return false; {5, 10, 19}
remove(19)	{5, 10, 19}	{5, 10}
remove(1)	{5, 10}	{5, 10}
search(5)	{5, 10}	return true

Searching



1. Compute hash(key(data)).
   Returns an index between 0 and size-1.

2. Read array[index].
   Must be able to distinguish "empty" from "non-empty" array element.

3. If empty, return false.

4. If non-empty, compare array element to key. If the same, return true; else return false.

Hash Table holds {231110, 59343, 67335, 122117}

Why is searching a hash table fast?

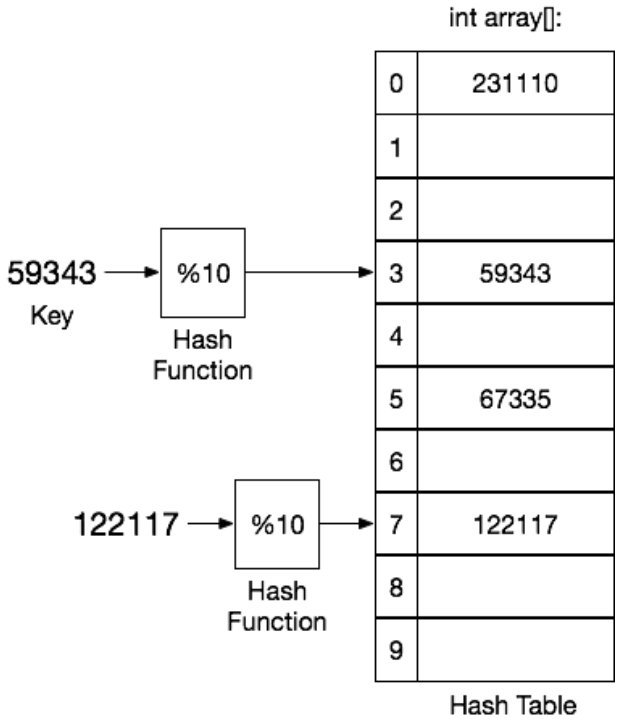
We can calculate an index in O(1) modulo arithmetic

We can lookup an array entry in O(1) array indexing

We can compare keys for a match in O(1) integer comparison

Hence, search can be done O(1)

Insertion



1. Compute hash(key(data)).
   Returns an index between 0 and size-1.

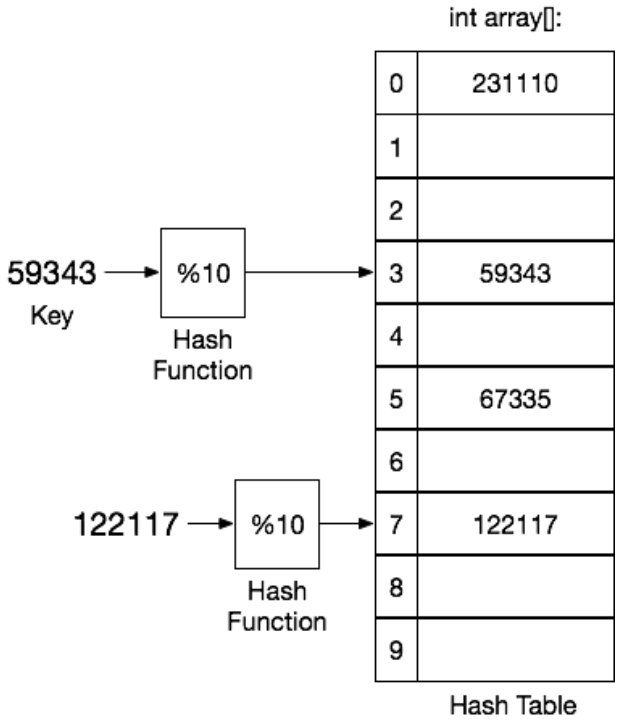
2. Read array[index].

3. If empty, store data in array[index]
   and return true.

4. If non-empty, this is a collision.
   For now, return false.

Hash Table holds {231110, 59343, 67335, 122117}

Removal



1. Compute hash(key(data)).
   Returns an index between 0 and size-1.

2. Read array[index].

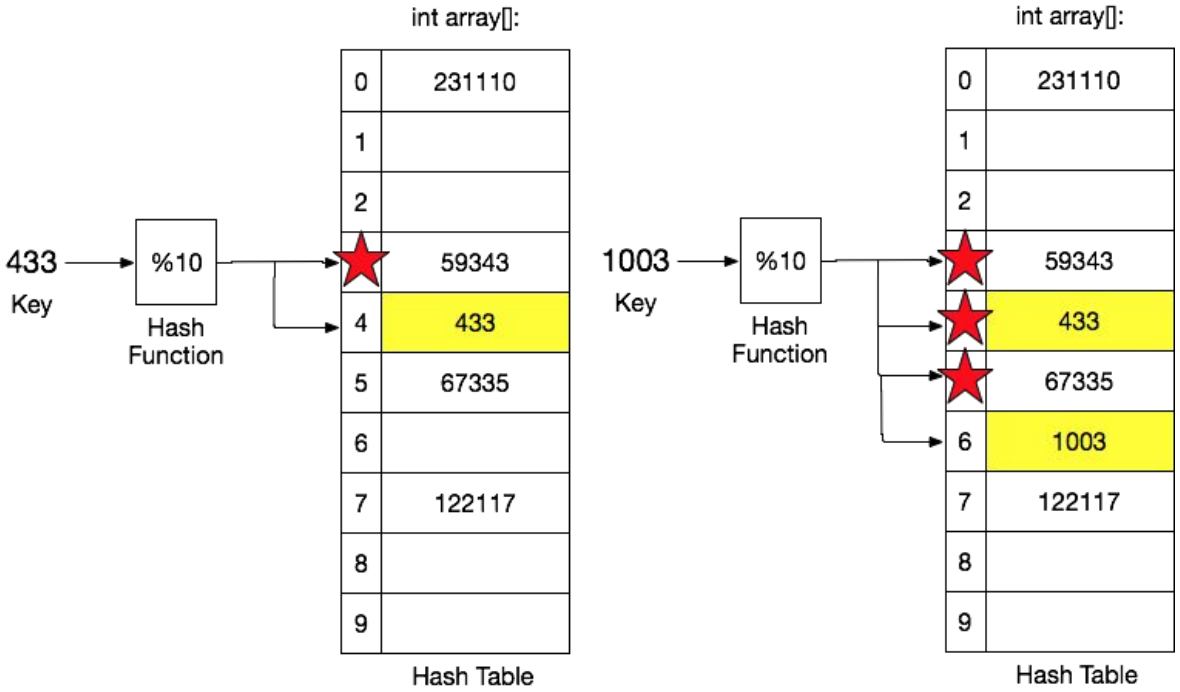
3. If empty, return.

4. If non-empty AND array[index] == data,
   set to empty and return.

Hash Table holds {231110, 59343, 67335, 122117}

Linear probing to handle collisions

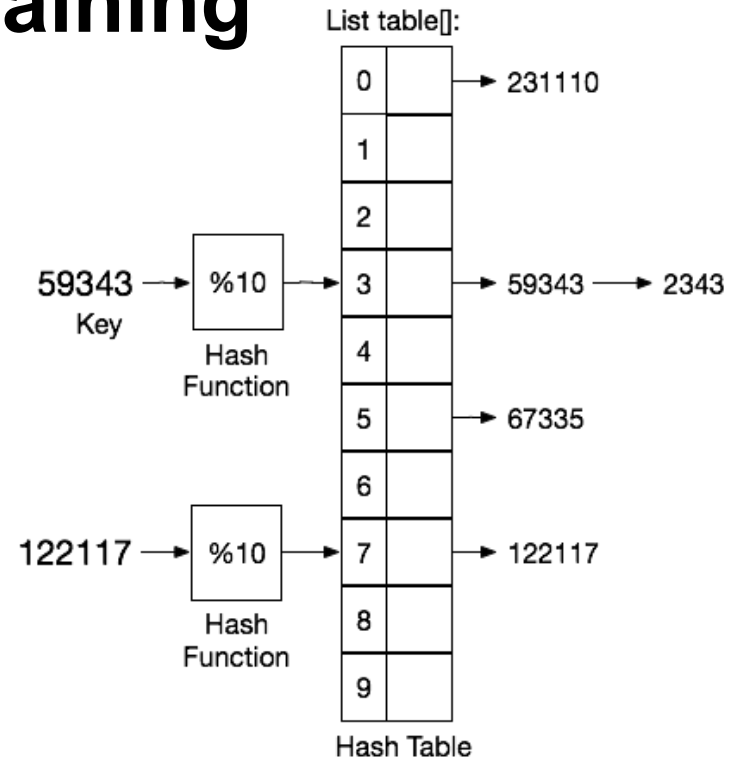
If the default entry is occupied, go to the next empty index and use it instead



Approach 2: Chaining

Chain all of the data in the same bucket together in a linked list.

The table is now an **array of linked lists**.



Linear probing versus Chaining

- Linear probing is faster when you know the amount of data in advance:
  - It uses empty space within the hash table efficiently
    - Doesn't need to allocate additional memory as long as collisions are infrequent.
    - But, number of hashed items ultimately limited by hash table size.
  - Quick search of adjacent buckets (i.e. no list traversal)
  - But, more sensitive to the quality of the hash function
    - It's possible that we run out of buckets within the probeDistance: (1) due to collisions on the same bucket, and (2) hashing to adjacent buckets

Linear probing versus Chaining

- Chaining tolerates more uncertainty in the volume of data
  - Insertion is always fast, because items are pushed onto a list in O(1)
    - We can always add items to a chain; not limited by hash table size.
  - Search can be slower (list traversal), but most chains have 0 or 1 item
  - Collisions degrade the efficiency of search
    - If chains become long, the table needs to be made bigger and all elements hashed to new locations

Other important topics for hash tables

Rather than linear probing, we could use multiple hash functions to find an empty location

- Sometimes used in hardware caches (ECE 463)

There are many common hash functions (ZyBook 12.7), but two important classes:

- string hash functions, integral hash functions (applicable to pointers)
- It's important to make sure they are computed in O(1)

Hash function

A ***perfect hash function*** would spread keys uniformly across buckets, avoiding collisions.

- Perfect hash functions don't exist, in general.
  Can create if we know the size and all possible keys in advance.

- But, we use them as a comparison point for analysis

Hash function

Modulo hash functions use the remainder after dividing by the size of the table (forces indices in proper range)

index = key % size

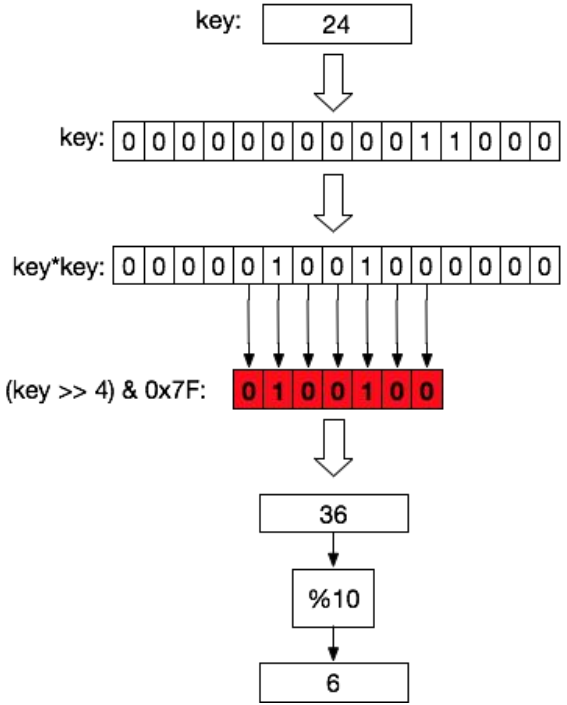
This appears good at first, but can cause problems if the keys share a common factor (e.g. 100, 200, 300 all hash to 0 with modulo 100)

Mid-square integer hash function

Idea: Randomize the bits used for the modulo operation

1. Square the key (key*key)
2. Extract the middle N bits, where N >= $\log_2$(size)
3. Modulo the result with the size of the hash table

Many others out there. Search “integer hash function”



String hash function

Daniel J. Bernstein created a popular string hash function:

```
int djb2(const char * str, int size) {
  int val = 5381; // randomize starting point
  for(int i=0; str[i]!='\0'; i++)
      val = val*33 + str[i]; // accumulate characters into a single value
  return val % size;
}
```

What about hashing other kinds of objects?

Objects are made up of fields that are integers, pointers, strings, etc.

Pick one field that uniquely identifies the object (i.e. name) and use that as the hash key

Based on its type, select a good hash function

If no single field is unique, multiply or concatenate fields together to form a key

- Example: Neither first name or last name is very unique, so concatenate first name and last name to make the key

std::hash

Standard C++ library provides a hash function that handles many data types.

```
size_t k1 = std::hash(89708);
size_t k2 = std::hash("The quick brown fox");
size_t k3 = std::hash(3.14159);
```

Actually, it's a template class (struct) that implements the function call operator...

Properties of Hash Table

For best performance, # buckets should be much **larger** than # items.
(Note: related to # items, not the number of possible keys.)

To avoid collisions, # buckets should be **prime**.

Hash function should be as **random** as possible -- map each key to a random bucket. Hash function should *uniformly* distribute keys among buckets.

Hashing depends on size of table: index = 0...size-1. If you are using std::hash, you need to reduce to the desired range using modulo (h % size).

Data is **unordered**: knowing where key N is located doesn't give any information about where N+1 might be.

When table starts to fill up, can **resize** to make it larger. How?