

Prof. Darrell Long  
CSE13

# Hashing and Bloom Filters




# Hash?

- Is it a food?
  - Yes.
- But for us, it means chopping things up and mixing them around.
  - Yes, like the food.
- We take a key (a string) and make a number from it.
- The resulting number should be *uniformly distributed*.



# Problem

- 
- We want to use this to find records based on a key.
  - Queries we should be able to make efficiently:
    1. Insert a key and its corresponding value.
    2. Search for a key and fetch the corresponding value.
    3. Delete a key along with its corresponding value.

# Data Structures that can be used to solve the problem?

- Expected time in arrays and linked lists:
  - Linear search: look-ups in unsorted arrays take  $O(n)$ .
  - Binary search: look-ups in sorted arrays take  $O(\log(n))$ .
  - Linked list look-ups take  $O(n)$ .
- Direct address table is a key-value store where the key is used as an index to the value :
  - Random access arrays: searching in in  $O(1)$  time.
  - Limitations:
    - Need prior knowledge of maximum key value.
    - Wastage of memory space if there is a significant difference between total records and preset maximum table size.

# Hashing

- The idea behind hashing or scatter storage is to scramble up aspects of the key and use this partial information for searching.
- Improvement over Direct Address Table:
  - Uses a function that hashes a larger fixed key into a smaller number:
    - Less than or equal to the size of the table.
  - Uses the smaller number as index of the value.
- With hashing we get  $O(1)$  search time on average (under reasonable assumptions) and  $O(n)$  in worst case.

# Hash Tables

- An unordered collection of key-value pairs.
- Offers efficient lookup, insert, and delete operations:
  - Neither arrays nor linked lists achieve all of these.
- The Python dictionary is a hash table. Example:
  - `my_dict = {1:'Ron', 2:'Bob'}`
  - Key 1 is associated with the name 'Ron' and key 2 is associated with the name 'Bob'.

# Hash functions

- A function that hashes keys to generate indices for each value:
  - For a hash table with  $m$  slots:
    - $h(key) \rightarrow index \in [0, m - 1]$
- Characteristics of a good hash function are determined by the data being hashed:
  - Function uses all input data.
  - Computation is fast.
  - Minimizes hash collisions:
    - Uniformly distributes data across entire set of possible hash values.
    - Generates very different hash values for similar strings.
- Types of hash functions
  - Division based.
  - Multiplication based.
  - String-valued keys.

# Hash functions: Division based method

- Most simple method for hashing is the remainder modulo method.
- Formula:  $h(K) = K \bmod M$ 
  - $K$  is the key.
  - $M$  is the table size.



# Hash functions: String-valued key

- Sometimes you want to use string-valued keys for a hash table.
- Basic approach:
  - Take characters from string-valued key.
  - Compute an integer using an appropriate method.
  - Integer modulo the table size.
- Can we just add the letters?
  - Addition *commutes*.
  - No, the number of possibilities is too small.
- Can we just multiply them?
  - No, multiplication also *commutes*.
- Creating a *good* hash function is not easy.

# Example of a Hash Table

- Begin with an empty hash table.
- Store numbers with their respective keys:  
 $\{91:54, 38:26, 47:93, 56:17, 2:77, 82:31\}$
- Hash function  $h(key) = key \pmod{11}$ 
  - Remainder modulo method: key mod table size.
  - Remainder returned is an index whose value is between 0–10.
  - Values are stored in the index corresponding to each key.

0	1	2	3	4	5	6	7	8	9	10
None	None	None	None	None	None	None	None	None	None	None

$(h(item)=item\%11)$

0	1	2	3	4	5	6	7	8	9	10
77	None	None	None	26	93	17	None	None	31	54

## Load Factor ( $\lambda$ )

- Number of slots filled out of total slots in a hash table.
  - In this example, it is denoted by  $\lambda = \frac{6}{11}$ .

0	1	2	3	4	5	6	7	8	9	10
None	None	None	None	None	None	None	None	None	None	None

$$(h(item) = item \% 11)$$

0	1	2	3	4	5	6	7	8	9	10
77	None	None	None	26	93	17	None	None	31	54

# Birthday Paradox

- Let's say you are in a room with just 23 people, there's a 50-50 chance of at least 2 people having the same birthday.
  - Counting pairs: with 23 people we have  $\binom{23}{2} = 253$  pairs
  - The chance of 2 people having different birthdays is  $1 - \frac{1}{365} = \frac{364}{365} = 0.997260$ .
  - The probability of *all* 253 pairs having different birthdays is  $p = \left(\frac{364}{365}\right)^{253} = 0.4995$ .
  - Therefore, the probability that we find a pair with the same birthday is  $1 - 0.4995 = 0.5005$ .
- If you change the number of people ( $n$ ) in the room the probability changes such that

$$p(n) = 1 - \left(\frac{364}{365}\right)^{n \times \frac{n-1}{2}}$$

- This applies to hash functions as well.

# Hash Collisions

- Occurs when two pieces of data have the same hash value.
- Assume that  $h(x)$  is a hash function:
  - We have 2 keys  $K_i$  and  $K_j$  such that  $K_i \neq K_j$ .
  - A collision means  $h(K_i) = h(K_j)$ .
- Good hash functions, given a proper range of possible hash values, should rarely result in collisions.
- Collisions are unavoidable if the number of keys is greater than the number of hash values.
  - The cardinality of the domain of the hash function is larger than its range.



# Tackling Collisions

- Open Addressing allows elements to move from their preferred position to other positions:
  - Linear probing
  - Quadratic probing
  - Double Hashing
- Chaining using linked lists



# Linear Probing

- Move sequentially through the hash table slots until you encounter the next empty slot.
- For a table  $t$  of size  $SIZE$ , if we want to store an element  $x$  with hash index  $i$  at location of  $t[i]$ , and that position is already occupied:
  - Try to store it in location  $t[(i + 1) \% SIZE]$ , where  $index = (i + 1) \% SIZE$
  - If that position is occupied, we try  $t[(i + 2) \% SIZE]$  and so on.
  - Until we find an  $index$ , such that either  $t[index] = x$ , or  $t[index] = NULL$ .
    - This is for the `find()`, `add()`, and the `remove()` functions.



# Quadratic Probing

- Square the sequence of hash values when deciding how far to look for the next slot.
  - Do this if the initial probe fails.
- From the previous code: if we want to store an element  $x$  with hash index  $i$  in the table location of  $t[i]$  and that position is already occupied:
  - Try to store it in location  $t[(i \times i) \% SIZE]$
  - If that position is occupied, we try  $t[(i + 1) \times (i + 1) \% SIZE]$  and so on.
- To guarantee that quadratic probing hits every single available spot eventually:
  - Hash table size must be a prime number.
  - Hash table must never be more than half full.



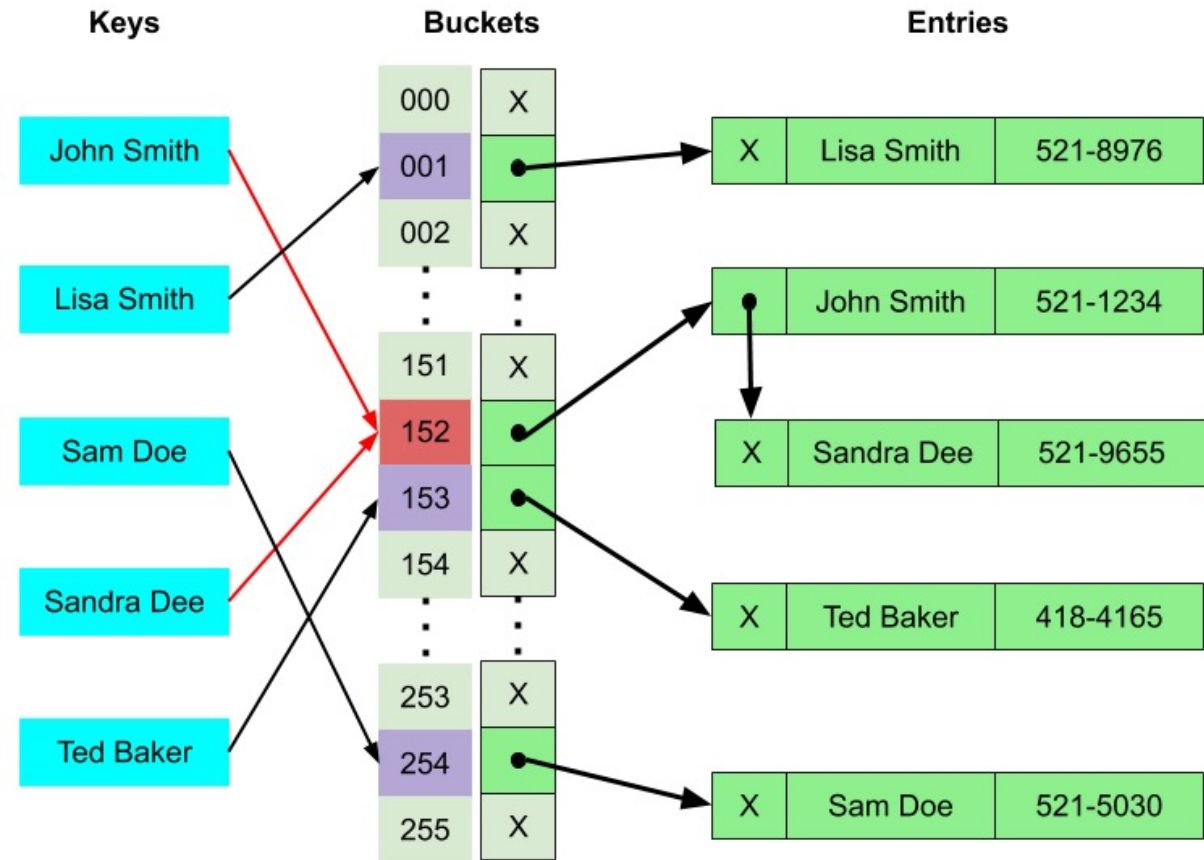


# Double Hashing

- Like Linear and Quadratic Probing.
- Uses a hash function to determine the  $(i + 1)^{st}$  hash location in table  $T$  containing  $n$  elements.
- Load Factor  $\lambda = \frac{n}{|T|}$
- For example given hash functions  $h_1, h_2$  for a *key*:
  - First, we check the location of  $h_1(key)$ .
  - If its not empty, we calculate  $h_2(key)$ .
  - $T(i, key) = (h_1(key) + i \times h_2(key)) \bmod |T|$

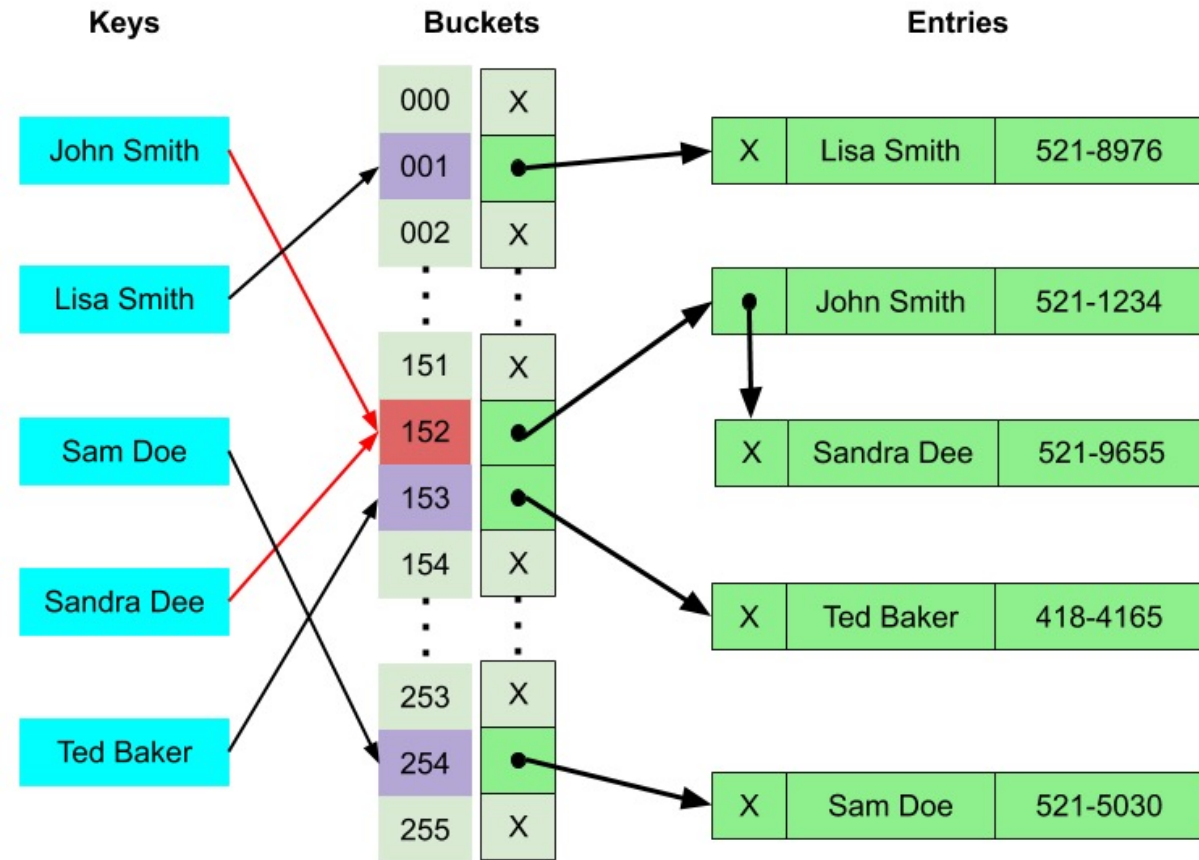
# Chaining using Linked Lists

- Each slot contains a link to a singly linked list:
  - Key-value pairs with the same hash.
- Look-up algorithm searches through the list to find matching key.
- Initially table slots contain nulls.
- List is being created, when value with the certain hash is added for the first time.



# Performance analysis of Chaining

- $m$  = Number of slots in hash table
- $n$  = Number of keys to be inserted in hash table
- Load factor  $\lambda = n/m$
- Expected time to search =  $O(1 + \lambda)$
- Expected time to insert/delete =  $O(1 + \lambda)$
- Time complexity of search insert and delete is  $O(1)$  if  $\alpha$  is  $O(1)$ .



# Open Addressing vs. Chaining

- Open addressing is computationally intense compared to chaining.
- Dynamic nature: in chaining hash table is never filled up as we can add more elements to the chain.
- Chaining is less sensitive to load factor and hash function.
- Cache Performance:
  - Chaining: bad as keys are stored using linked list.
  - Open Addressing: good as everything stored in the same table.
- Space:
  - Chaining uses extra space for links and parts of the hash table is never used.
  - No links in open addressing and a slot can be used even if an input isn't mapped to it.

```
typedef void* KeyType;
typedef void* InfoType;

#define EMPTY_KEY NULL
#define SIZE 11

typedef struct {
    InfoType value;
    KeyType key;
}Data_Item;

Data_Item* hash_table[SIZE];
```

Figure 1

```
#define SIZE 11

typedef struct {
    uint32_t value;
    uint32_t key;
}Data_Item;

Data_Item* hash_table[SIZE];

uint32_t hash_code(uint32_t key) {
    return key % SIZE;
}
```

Figure 2

# The Hash Table ADT

---

- Figure 1 shows a general hash table ADT:
  - KeyType and InfoType are void \* type so we can pass any type we need for our problem.
  - Based on the type of key, the hash function hash\_code() will change.
- Figure 2 shows a hash table declaration for the problem on slide 12:
  - Hash table of size 11: as we discussed before the hash table size should be a prime number.
  - Both the value and the key are unsigned 32-bit integers.
  - The hash function used here is the remainder method that takes a 32-bit integer as a key and returns an index in the table between 0-10.
- Each data item consists of a key-value pair.

```
void display() {  
    int i = 0;  
    for(i = 0; i < SIZE; i++) {  
        if(hash_table[i] != NULL)  
            printf("(%d,%d)\n", hash_table[i]->key, hash_table[i]->value);  
        else  
            printf("NULL\n");  
    }  
}
```

© 2020 Darrell Long

## Displaying a Hash Table

```
Data_Item *search(uint32_t key) {  
    uint32_t index = hash_code(key);  
    while(hash_table[index] != NULL) {  
        if(hash_table[index]->key == key)  
            return hash_table[index];  
        ++index;  
        index %= SIZE; // Linear Probing  
    }  
    return NULL;  
}
```

## Search in a Hash Table

- Searching using Linear probing to avoid hash collisions:
  - When an *index* is used, you will check the next index *index + 1*.
  - This type of search wraps around the hash table and continues from the start of the array.

```
void insert(uint32_t key, uint32_t data) {  
    Data_Item *item = (Data_Item*)calloc(1, sizeof(Data_Item));  
    item->value = data;  
    item->key = key;  
    uint32_t index = hash_code(key);  
    while(hash_table[index] != NULL) {  
        ++index;  
        index %= SIZE;  
    }  
    hash_table[index] = item;  
}
```

## Insertion

- Since traversing the Hash Table is done using linear probing:
- Sequentially move forward through the indexes.



```
void delete(Data_Item* item) {  
    uint32_t key = item->key;  
    uint32_t index = hash_code(key);  
    while(hash_table[index] != NULL) {  
        if(hash_table[index]->key == key) {  
            hash_table[index] = NULL;  
            return;  
        }  
        ++index; Danger!  
        index %= SIZE;  
    }  
}
```

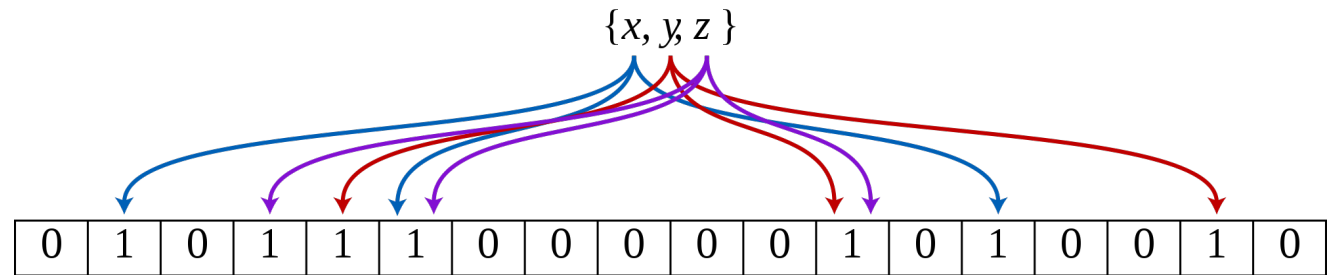
- This function takes as input a key value pair and if the contents of a hash table index matches the key, deletes the corresponding item.
- Traverses forward through the hash table using linear probing.

## Deletion

# Bloom Filters

- A data structure designed to tell you whether an element is present in a set.
- Internally structured like a bit array.
- Two operations: Add and Search/Query.
- Assume bloom filter on the right has  $k = 3$  hash functions:
- Pass elements  $\{x, y, z\}$  through 3 hash functions and set the corresponding bits in the bloom filter.

Assume that the  $K$  hash functions are  $h_1, h_2$ , and  $h_3$ .



$h_1(x), h_2(x)$ , and  $h_3(x)$  sets bits 1, 5 and 13.

$h_1(y), h_2(y)$ , and  $h_3(y)$  sets bits 4, 11 and 16.

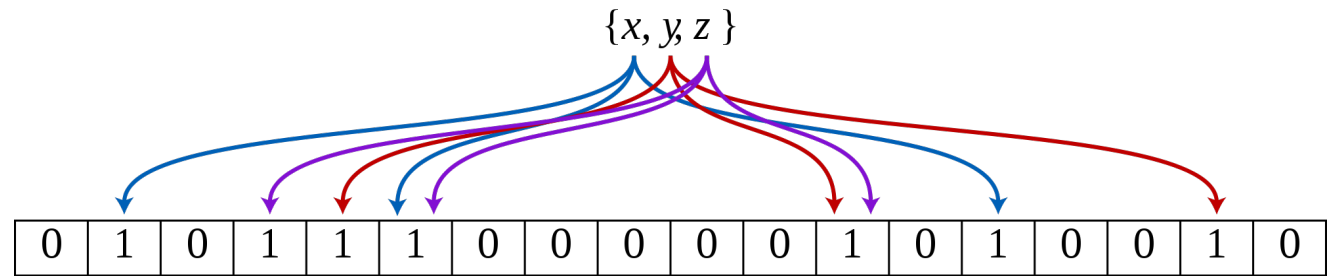
$h_1(z), h_2(z)$ , and  $h_3(z)$  sets bits 3, 5 and 11.

# Why Bloom Filters?

- Developed by Burton Howard Bloom in 1970.
- Bloom filters are fast and memory efficient.
- Called filters:
  - Used as a first pass to filter out segments of datasets that do not match a query.
- Related to hash tables:
  - Can be used to efficiently check if a key exists in the hash table.

## False Positives

- A bloom filter query returns “possibly in set” or “definitely not in set”.
- False positives:
  - When passed through the same 3 hash functions 2 bits are set by 2 different elements from the set.
  - Bit 5 was set by x and z, Bit 11 was set by y and z.



$h_1(x), h_2(x),$  and  $h_3(x)$  sets bits 1, 5 and 13.

$h_1(y), h_2(y),$  and  $h_3(y)$  sets bits 4, 11 and 16.

$h_1(z), h_2(z),$  and  $h_3(z)$  sets bits 3, 5 and 11.

# Probability of False Positives

- Assume that a hash function selects each array position with equal probability.
  - M is the number of bits in the array
  - K is the number of hash functions
  - N is the number of inserted elements.
- $p = (1 - e^{-\frac{K \times N}{M}})^K$
- The more items per Bloom filter, the higher the chances of having false positives.
- Increasing the number of hash functions can reduce false positives

# Time and Space Complexity

- Size of bloom filter is  $M$  bits and  $K$  hash functions:
  - Insertion and Search will both take  $O(k)$  time.
- The space of the actual data structure (what holds the data) is simply  $O(M)$ .

# Other application of hash functions: Cryptographic Hash

- Message Integrity:
  - Alice sends Bob a message,  $m \parallel h$  where the message is concatenated with the message's hash.
  - Bob computes the hash over the message and compares the received hash and the calculated hash.
  - If they are different, the message has been altered.
- Digital Signatures:
  - Alice signs a message with her private key.
  - Bob can verify Alice's authorship with her public key.
  - Inefficient to sign the entire message. Better to sign the message's hash value.

# Cryptography based Hash Function

- An ideal cryptographic hash function guarantees the following properties:
  - Deterministic: same message results in the same hash.
  - Quick to compute the hash for any given message.
  - Infeasible to construct a message that yields a given hash value.
  - Infeasible to find two different messages with the same hash value.
  - Avalanche effect — A small change to the message should produce a hash value that appears uncorrelated with the old hash.



# Popular cryptographic hash functions

- MD5 – broken
- SHA1 – broken
- SHA256
- SHA512
- SHA3

# Summary

- Hash functions
  - Map data of arbitrary size to fixed sized values.
  - Aim to avoid collisions
- Hash table
  - A data structure that maps keys to values
  - Provides  $O(1)$  access times on average.
  - Index into a hash table is calculated by a hash function
- Bloom Filters
  - Probabilistic data structure that tests whether an element is a member of a set.
  - The element is either not in the set or it may be in the set.