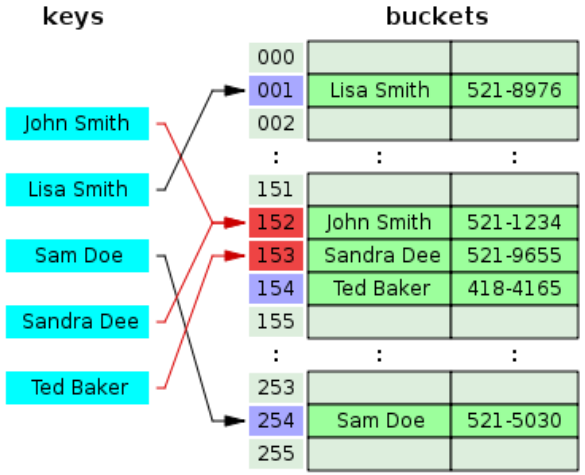
Hashing and Indexing
{Part 1: Hashing}

CS 5012



UNIVERSITY OF VIRGINIA
DATA SCIENCE
INSTITUTE

We thought we found the answer

- While studying linear search, we learned that we have to go through every entry one by one to find the desired item. Time complexity: **O(n)**
- It's *boring* and *time consuming!*
- When searching an ordered list, we thought we had found the answer… binary search
- In a sorted list, using binary search, the search is lightening fast (*in comparison!*) Time complexity: **O(log(n))**
- Increasing the size of the data, changed the search time *very slightly*

“Magical” Data Structure

- How about if we can find an item in an array, almost, *without* search
- We type our search key and our algorithm takes us directly to the item we are looking for
- No need to search through all the items: 0 to n-1
- Such a *magical* data structure ***DOES*** exist, it’s called a **Hash Table**

What is a Hash Table?

- A hash table (also *hash map*) is a data structure used to implement an associative array, a structure that can map keys to values

- A hash table uses a **hash function** to compute an **index** into an array of *buckets* or *slots*, from which the correct value can be found

What is a Hash Table?

Hash Table: Can be searched for an item in

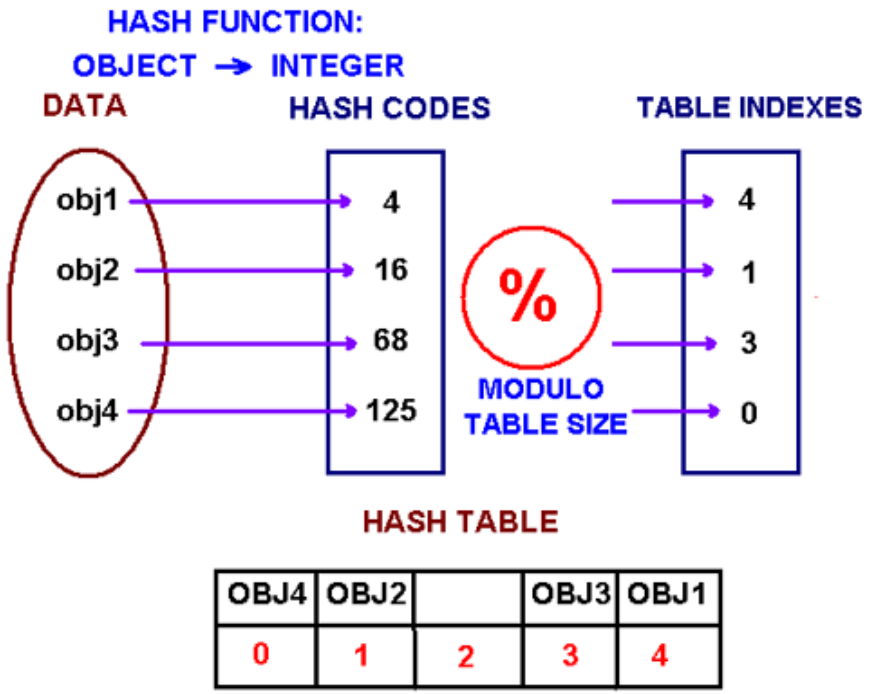
O(1) time

using a hash function to form an address from the key

- Creating a hash table has two distinct operations:
  1. **Creating an array**: The actual table where the data is stored
  2. **Creating the Hash Function**

What is a Hash Function?

- **Hash Function:** Function which, when applied to the key, produces an integer which can be used as an address in a hash table (index into an array of *buckets*)



What is a Hash Index?

- **Hash Function:** Function which, when applied to the key, produces an integer which can be used as an address in a hash table (index into an array of *buckets*)

- **Hash Index:** A hash index organizes the search keys with their associated pointers into a hash file. It consists of a collection of buckets organized in an array. Through linked lists, multiple items can be associated with one index because of this Hash indices are often called *buckets*

Hashing Basics

- A **uniform hash:** when the indices produced by the hash function (into an array) are equally likely to be generated

- *Ideally*, the hash function will assign each key to a unique bucket (index), but this ideal situation is rarely achievable in practice (unless the hash keys are fixed; i.e. new entries are never added to the table after it is created)

- Instead, most hash table designs assume that **hash collisions**—different keys that are assigned by the hash function to the same bucket—will occur and must be accommodated in some way

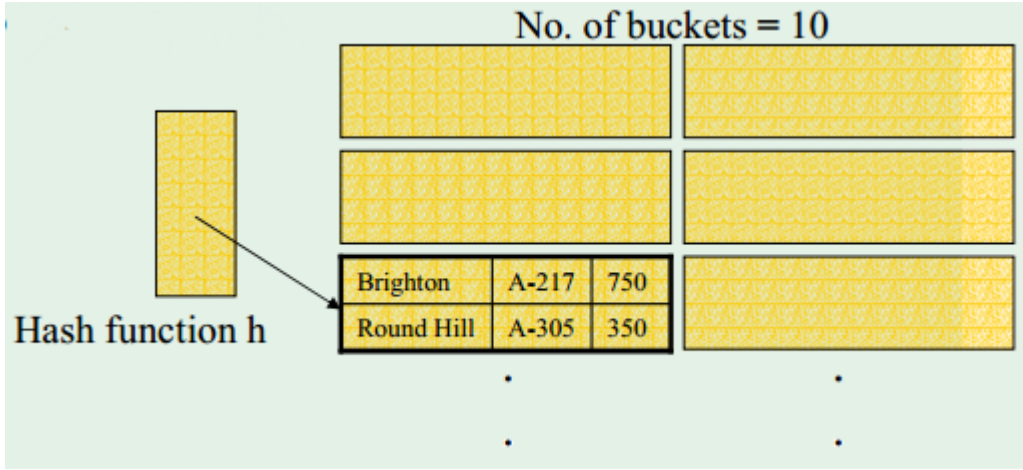
Hashing Basics

- In a well-dimensioned hash table, the average cost (*number of instructions*) for each lookup is ***independent*** of the number of elements stored in the table
- In many situations, hash tables turn out to be more efficient than **search trees** or any other table lookup structure

  For this reason, they are **widely used** in many kinds of computer software, particularly for associative arrays, database indexing, caches, and sets
- **Key-Value** pairs: the value portion need not be a single item Can be: `CID,{FN, LN, Age, Major, Year, …}`

Hash Index

- Puts a pointer to a record in the bucket based on the hash of that particular record's key
- A ***hash function*** "***h***" maps a search-key value to an address of a bucket
- Commonly used hash function: ***hash value mod*** n_B where n_B is the number of buckets
- E.g. ***h***(Brighton) = (2+18+9+7+8+20+15+14)mod 10
  = 93 mod 10 = 3
  [**bucket #3**]



Hash Function

- The values returned by a hash function are called **hash values**, **hash codes**, hash sums, or hash integers

- Implementation of a hash function is in two steps:

1. Creating the original hash code for the hash key
2. Transferring this number to a smaller number to be used as index for the key

Hash Function

- In addition to developing hash tables, hash functions have numerous other applications:
  - Used to build caches for large data sets stored in slow media such as a slow Hard Drive. Frequently accessed data is saved in cache and can be accessed in a very short time
  - Computer operating systems use hash functions for fast access of key words and commands
  - There are several other applications…

A Good Hash Function

- The very basic property of a hash function is the ability to transform the hash keys in such a way that all the data pairs have a hash code in the **range of 0 to <= m**, where **m** is the size of the hash table
- When hashing **strings**, it's important to consider *all characters* so that the probability of getting a unique code is enhanced

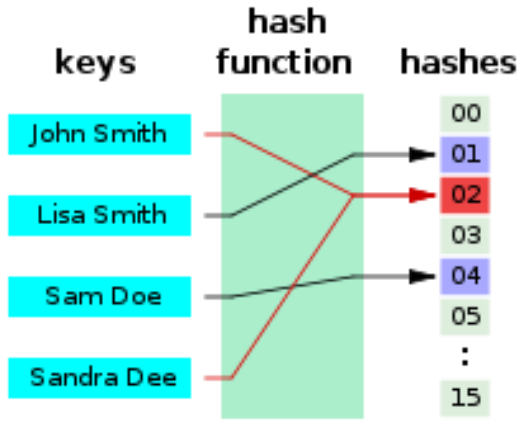
A good hash function is very unlikely to
return the same code for *unequal* keys but
it ***must return the same code for equal keys***

A Good Hash Function

- **A good hash function** should create hash codes that permits the widest distribution of keys to various index values of the hash table. (*It should uniformly distribute the keys*)
- A **prefect** hash function will create such a unique code for each key in the data that the probability of two keys having the same code is zero. Such a hash function is ideal but rarely does it exist
- There is always a possibility of several keys having the same hash code. It causes a **collision**

Collision

- When 2 keys hash to the **same location** in the hash table a **collision** occurs. There are some techniques available to resolve the problem of collisions
- It is to be noted that a good hash function distributes the elements uniformly over the hash table, this keeps the number of collisions within manageable limits
- There are two basic methods for handling collisions in the hash tables
  1. Chaining
  2. Linear Probing

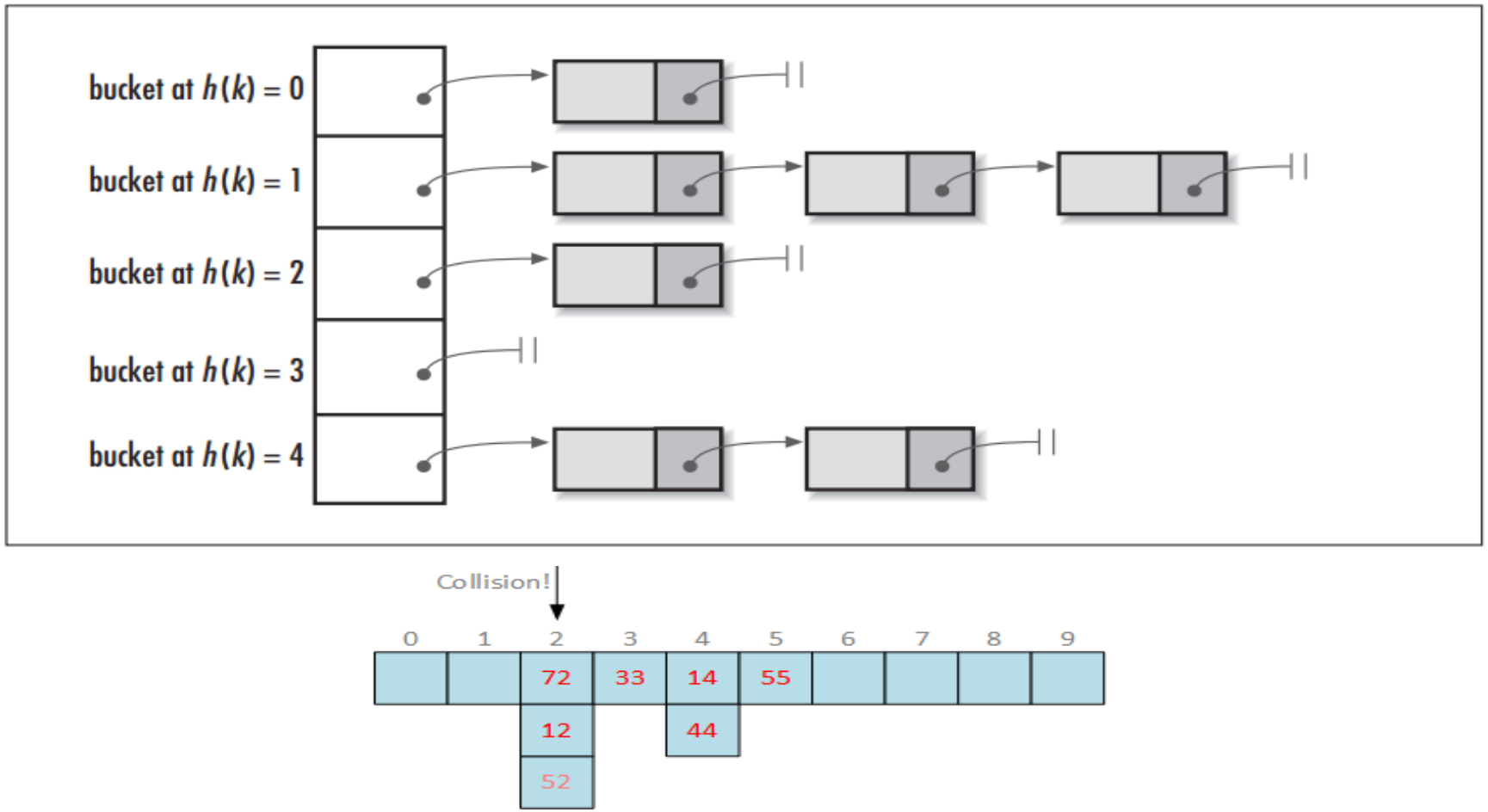


Collision Resolution: Chaining

- When there is a collision (key hashed to an **occupied slot**), the new key is stored in an ***overflow list***. Each incoming duplicate entry is linked at the end of this list chained to the index

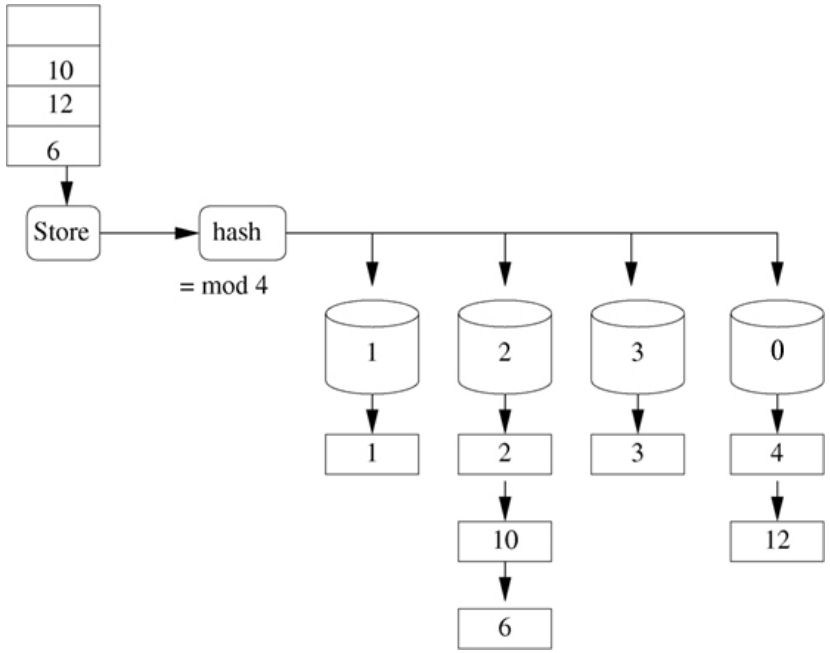
- One way of keeping the number of such colliding keys to a minimum is to make the **size of the hash table (*m*) <u>larger</u>** than the number of key value pairs.
  One rule of thumb is to make ***m*** about 33% bigger than the number of keys (***n***)

Collision Resolution: Chaining



Collision Resolution: Chaining

- Example of a simple hash function, where the bucket number equals the primary key. Number of buckets = 4

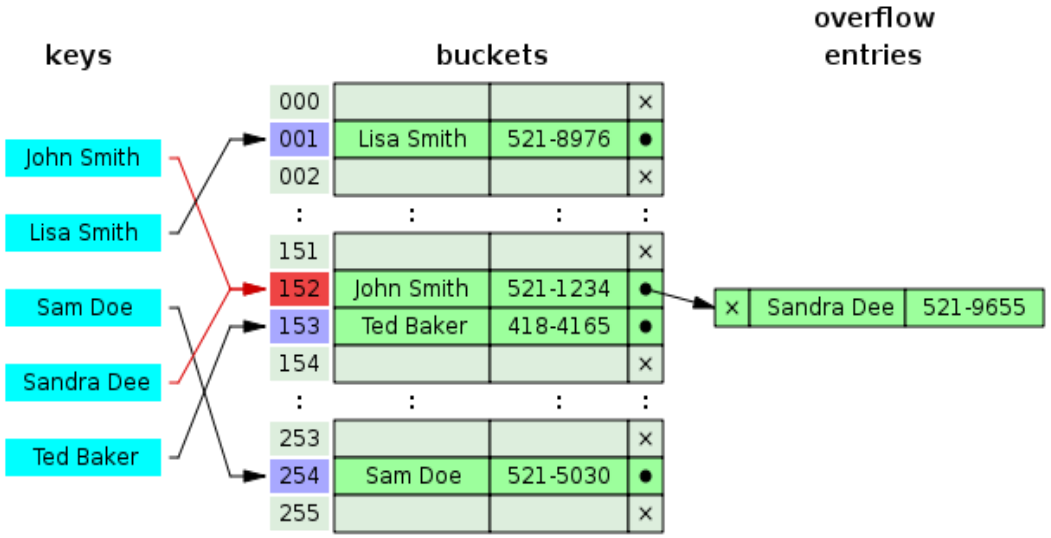


The linked lists can be seen at the bottom

Collision Resolution: Linear Probing

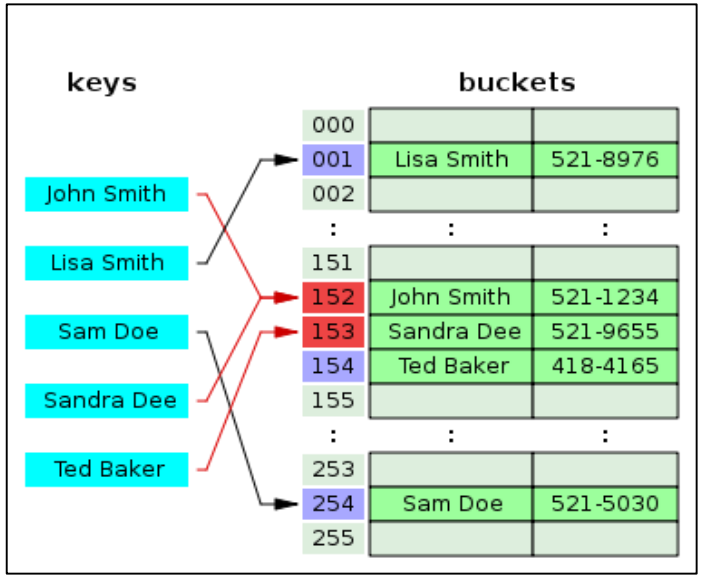
- If faced with a collision situation, the **linear probing** table will look onto to subsequent hash elements until the first free space is found
- Since the hash table goes in a linear fashion to look for an empty slot, it is called *linear probing*
- Obviously, the size of the hash table (***m***) must be bigger than the number of elements so that some empty slots are available

Example of Hash Collision Resolution



Hash collision resolved by open addressing with linear probing (*interval*=1). Note that "Ted Baker" has a unique hash, but nevertheless collided with "Sandra Dee", that had previously collided with "John Smith"

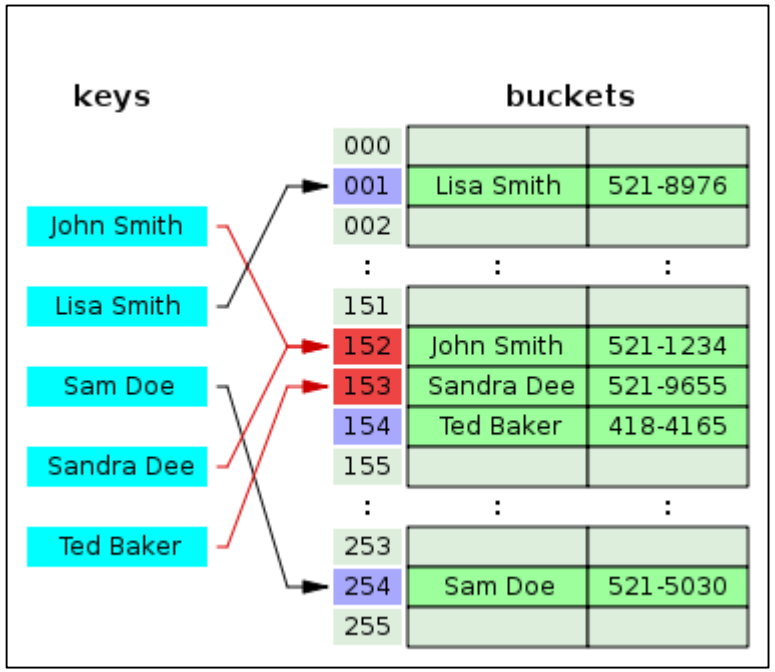
Hash collision resolved by separate chaining with head records in the bucket array



Collision Resolution: Linear Probing

- Unique to linear probing, is the inclusion of “flags”
- Each bucket is labeled (or flagged) with one of the following:
  - {E} – empty
  - {0} – occupied
  - {D} – deleted

- *How are these flags used?*



Hashing with Databases

- Hashing is of interest based on its usage in a **Database** – how it is implemented
- Certain DBs do not allow hash indices
  - Oracle does not have hash indices built in
  - **MySQL** does, but uses it in a weird way…
    - If you change a table's DB Engine to "**memory**" – it will automatically create a hash index for very fast lookups
    - Why would you use memory? This results in the entire DB being stored only in RAM, never written to disk. Only thing that is stored on disk is the .frm file (*DB schema*)
    - It speeds things up because forces nothing to be written back to disk