CSC2001F: Data Structures II

Omowunmi Isafiade

Email: omowunmi.isafiade@uct.ac.za

Office: Room 306

What we will cover in this block...

- Hash Tables
  - Linear Probing
  - Quadratic Probing
  - Double Hashing
  - Chaining
- Priority Queues
  - Binary Heaps
  - Heap sort
  - Merging
- Graphs
  - Graph Algorithms (Dijkstra, Bellman-Ford...)

Reference Textbook:

“Data Structures & Problem Solving using Java”, 4th Ed., Mark A. Weiss.

Hashing & Hash Tables: Outline

- What Hashing & Hash Tables are?
- Why Hash Tables are useful?
- Selecting a good hash function
- Methods of creating hash functions
- Summary

Overview: Hash Table Data Structure

- Purpose:
  - To support insertion, search and deletion operations in average-case constant time.
- Assumption:
  - Order of elements is irrelevant
  - Data structure “not” useful if you want to maintain & retrieve some kind of an order of elements (prioritizing access)
- The implementation of **Hash Tables** to perform insertion, search and deletion **operations** is called **Hashing**
- **Hash Function:** Hash[“String key”] => integer value (**index**)

Hashing: Motivation

Comparison to Binary Search Trees

The main attraction for considering hash tables include:

- Speed up of search operations:
  - Consider searching an array for a given value (Unsorted? Sorted?)
  - Knowing the **index** in advance (**hash function**)
    - Hash[“String key”] => integer value (**index**)
  - For instance in an array of 500 items, knowing the exact position of a specific element means we can access it directly without having to do a sequential search through each slot
- Thus, average search time for an element in a hash table is O(1) time.

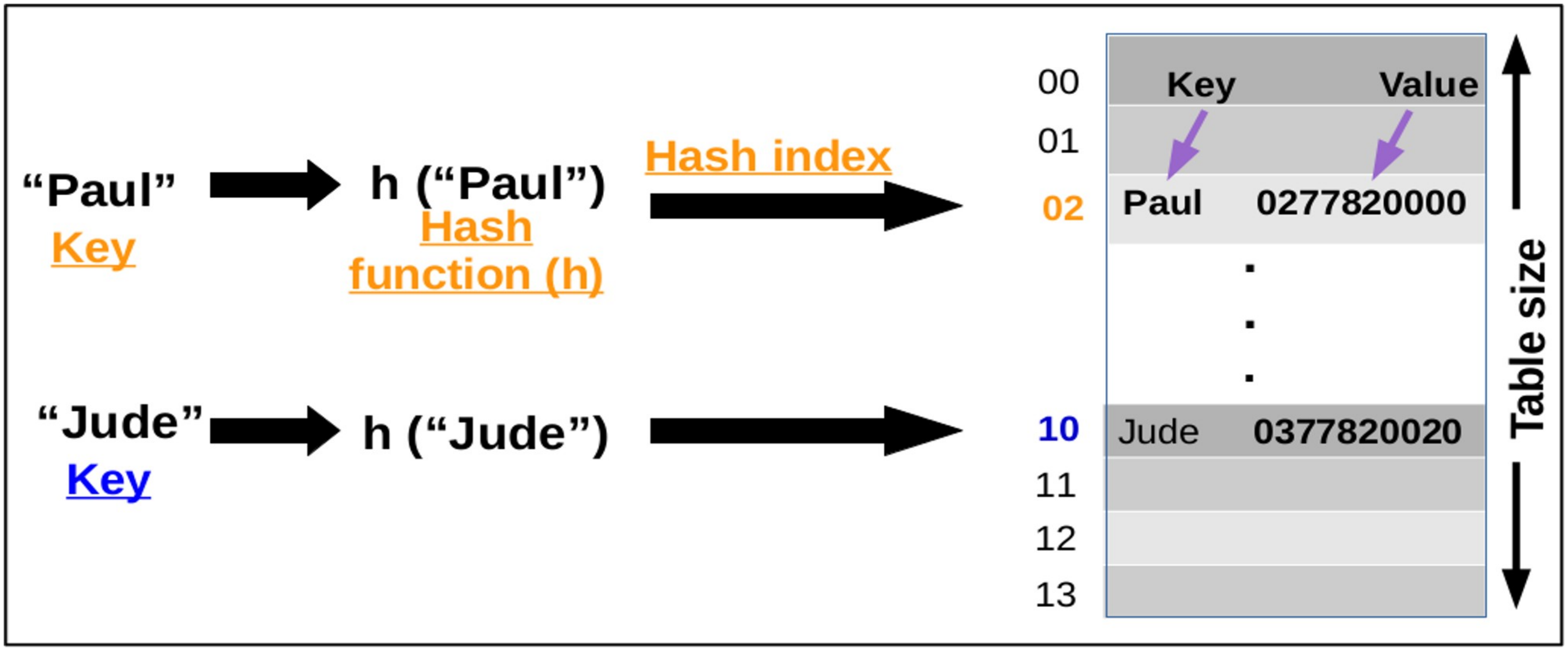
What a Hash Table is...

- A **hash table** (**hash map**) is a data structure that uses a **hash function** to map identifying values known as **keys** to their **associated values (constant time per operation)**.

- **A hash function, h(k),** converts the **key** into an integer suitable to index an array (of buckets or slots, ***m***), where the **value associated** with the **key** can be found.
  - $$h(k): U \rightarrow \{0, 1, \cdots, m-1\}$$

- **Example**: A key (e.g. a person's name or ID number) can be mapped to a corresponding value (e.g. a telephone number).

Hash Table: Essential Components



- A **hash function** basically **translates** the **key** (i.e. name of the user) into an **index** that **uniquely identifies** the **associated value** (phone number). Note: table size (**m)** = 14.
- **Hash function? Table size?**

Why Hash Tables are Useful...

- Many applications require a data structure that facilitates insert, search, and delete operations. Examples:

Compilers:

- Perform translations to machine language by maintaining a **symbol table.**

- In the symbol table, the keys are arbitrary character strings and values are identifiers in the language.

- Typically, only insert and search operations are performed.

Why Hash Tables are Useful...

Password Lookup:

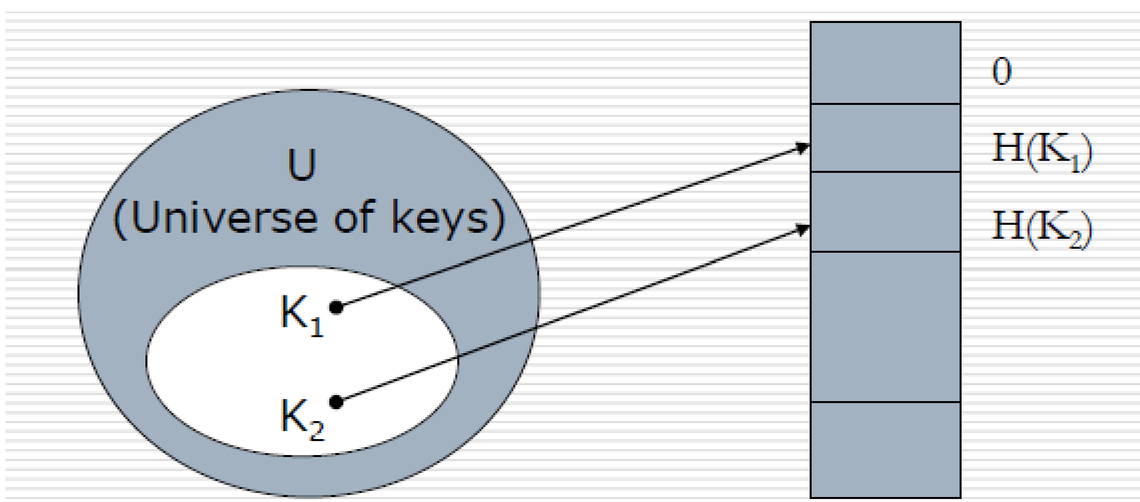
- In systems with multiple users
- Hash tables allow for a fast retrieval of the password which corresponds to a given username.
- Key (username and password) and Value (information associated with the user's profile)

Other application areas include:

- Spell Checkers
- Search Engines
- Game programs

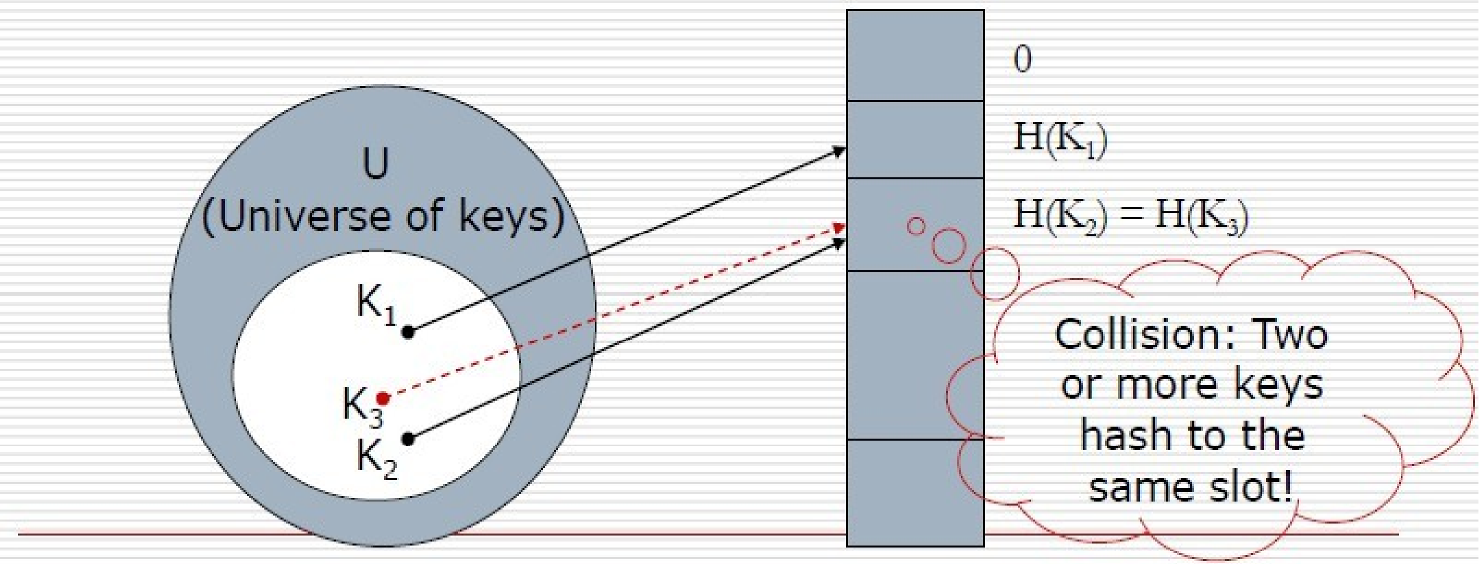
Designing Hash Functions

- Typically we aim to design hash functions as one-to-one functions to facilitate insert, search and delete operations.
- Try to avoid having two or more keys hashing to the same index value



Designing Hash Functions

- Two or more keys can hash out to the same position, causing a **collision**
- **Collision**: when $h(k_i) = h(k_j)\ for\ k_i, k_j \in U, \wedge k_i \neq k_j$
- But! Usually hash functions are designed as many-to-one functions in order to deal with collisions flexibly.



Using a hash function h to map keys to hash table slots

What makes a good hash function?

Desired properties of a hash function h(k):

- Simple and quick to compute
- Distributes keys (U) uniformly across the hash table
- Consistent in identifying associated values (map equal keys to same index)
- Minimizes the probability of collisions
  - Too many collisions result in poor performance (searching, inserting, deleting)

Designing a good hash function- some challenges

- Difficult to check that each key will hash to a unique slot.
- Requires checking every possibility (exhaustive search – difficult) or knowing the probability distribution from which the keys are drawn (hard)
- **Example**:
  - A key (e.g. a person's name) can be anything – not known beforehand.
  - Birthday paradox: estimate the odds of finding two people with the same birthday

Designing a good hash function-Method

- Possibly heuristics ???
  - Use information about the keys to decide on a good hash function
- Example:
  - Consider a password checker table in which the keys are character strings representing a user's profile
  - Closely related passwords like hary123 and harry123 can happen
- Heuristics → aiming to minimize the chance of collisions

Hash Function - From Keys to Indices

- h(k) basically has two components:
  - **Hash code map** (f)
  - **Compression map** (g)
- Function f maps the universe of keys U onto the integers. i.e. $f : U \rightarrow$ integers (I)
- Function g hashes the resulting integer.

$$g : I \rightarrow \{0,1,2,\ldots, m-1\}$$

- So, $h(k) = g(f(k))$,
- If k is a positive integer, then $f(k) = k$
- What happens when the keys are not integers?

From Keys to Indices

- Keys as set of integers
  - Most hash functions assume that the universe of keys will fall within the set of integers (e.g. 0, 1, 2,...)
  - Reason: makes search, delete and insert operations faster and more precise.
  - No need to consider fractional components (floating point) e.g, 2.3466
- When the keys are not natural numbers we try to find ways of translating the keys into integers

From Keys to Indices

- Dealing with hashing non-integer keys:

  Find ways of translating keys into integers. Example:

  - Remove hyphen in 7398-4605 → 73984605
  - String: add up ASCII values of the characters in the string (e.g. java.lang.String.hashcode())
  - Then use standard hash function on the integers
- Note: character can be expressed in **radix notation**

From Keys to Indices

- The mapping of keys to indices of a hash table is achieved using a hash function h(k), which usually comprises of two maps:
    - Hash code map: key → integer
    - Compression map: integer → [0, M-1]
- If your key is already an integer, no need for integer conversion (hash code map)
- A good hash function minimizes possibility of collisions
- M is the size of the array (so an index is a value between $0 \cdots M-1$)

From Keys to Indices: Hash Code Maps

- **Integer cast**: consider numeric types with 32 bits or less, we can reinterpret the bits of the key as an int.
- **Component sum**: for numeric types more than 32 bits (long or double), partition the bits of the key into components of fixed length of 32 bits and sum the components, ignoring overflows.
  - Note: not a good choice for strings – many collisions e.g, **teas**, **seat**)
- **Polynomial accumulation**: multiplication by a constant c makes room for each component in a tuple of values, while also preserving a characterization of the previous components (i.e radix notation)

From Keys to Indices: Radix Notation

- A character can be expressed in radix notation. So we can express the string "person" as the set of integers:
  - "person" → (112 101 114 115 111 110)
  - "junk" → (106, 117, 110, 107)
  - Where 112 is the ASCII notation for "p", 106 for "j"
  - Exercise: Express "person" as an integer value using a radix-2 notation.
  - Note: In a system with radix- $x, (x > 1)$ notation, a string of digits, $d_1 \ldots, d_n$ denotes the decimal number

$$d_1 x^{(n-1)} + d_2 x^{(n-2)} + \cdots + d_n x^0$$

From Keys to Indices: Dealing with Overflow

- Conversions from string to integer can create numbers that are too large to store
- Consequence: large arrays that make search/ delete/ insert operations cumbersome (expensive and slow)
- Solution: use a function that maps large numbers into smaller, more manageable ones (e.g Division method)

Hash Functions - Dealing with Overflow

- Several ways of creating hash functions that handle overflow (compression maps)
- We consider two: **Division method** and **Multiplication method**
- Division Method:
  - Map a key, *k,* into one of the slots m in the hash table by taking the remainder of *k* divided by m
  - Hence the hash function is: *h(k) = k mod m* where m is the table size
- Table is represented as a series of slots going from $0 \cdots m-1$

Hash Functions - Dealing with Overflow

- Note:
  - M is the size of the array (so an index is a value between $0 \cdots m-1$)
  - *K* is the key (integer value derived from the string conversion)

- Exercise: Using the hash function h(k) = k mod m, insert the strings “junk” and “person” into a hash table of size 11

- HINT: **Remember to evaluate the hash code map by representing the string using radix-2 notation first.

The Multiplication Method

- Operates in two steps:
  - Multiply the key *k* by a constant *A* in the range 0<A<1 and extract the fractional part of *KA*
  - Multiply *KA* by m (tableSize) and take the floor of the result
- Precisely the hash function is:

$$h(k) = \lfloor m(KA \, mod \, 1) \rfloor$$

- Where *KA* mod 1 means the fractional part of *KA,* i.e. $KA - \lfloor KA \rfloor$

The Multiplication Method

- Exercise: using the hash function $h(k) = \lfloor m(KA \bmod 1) \rfloor$ insert "person" and "junk" into a hash table of size 10
- Note: m is the size of the hash table, A = 0.3. Also remember we are using radix-2 notation to represent strings
- Recall: expressing a string of digits $d_1 \ldots, d_n$ in radix-2 notation is done as follows:

$$\rightarrow \quad d_1 x^{(n-1)} + d_2 x^{(n-2)} + \cdots + d_n x^0$$

- "person" $\rightarrow$ (112, 101, 114, 115, 111, 110) in ASCII
- "junk" (106, 117, 110, 107) in ASCII

Caching the hash ... (Example)

- Java 1.3 and beyond (avoiding expensive re-computation on same string)
- Caching the hash works because Strings are immutable (recall abstract data types...)

```
public final class String
{
      Public int hashCode( )
      {
            If (hash != 0)
                  return hash; // (2: previous result recalled)

            for (int i = 0;  i <  length ( );  i++)
                  hash = hash * 31 + (int) charAt( i );
            return hash;
      }

      Private int hash = 0;  (1: hash initialized to 0)
}
```