CS 2100: Data Structures & Algorithms 1

Hash Tables

Intro. To Hash Tables; Separate Chaining

Dr. Nada Basit // basit@virginia.edu

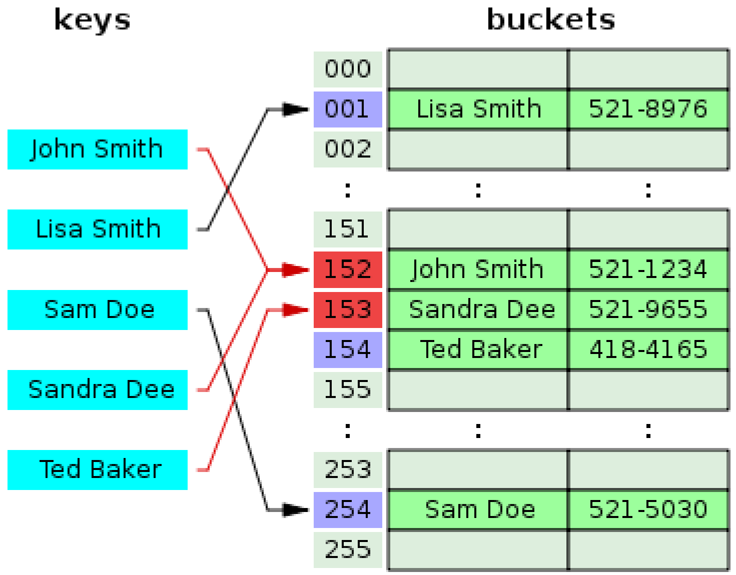
Spring 2022

Friendly Reminders

- The University updated the mask policy. As per my Request on Mar 28, 2022 (see Collab), I would greatly appreciate if you would do me a kind favor by **continuing to wear your masks** in CS 2100 (Ridley G008). I know it is a lot to ask, and it is **voluntary**, but I appreciate your understanding.
- If you forget your mask (or mask is lost/broken), I have a few available
  - Just come up to me at the start of class and ask!
- No eating or drinking in the classroom, please
- Our lectures will be **recorded** (see Collab) – please allow 24-48 hrs to post
- If you feel **unwell**, or think you are, please stay home
  - *We will work with you!*
  - At home: eye mask instead! Get some rest ☺

Hash Tables

An Introduction to Hash Tables



We Thought We Found The Answer

- Recall the linear search algorithm?
  - 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. What was more efficient? Binary search algorithm
  - In a sorted list, using binary search, the search is very 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?

- What if we could 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**

The Kind Of Data Is Stored?

- **Hash tables** store **key-value pairs**
  - Each **value** has a specific **key** associated with it
  - The value portion doesn't need to be a single item:
    - **Example:** `CID,{FN, LN, Age, Major, Year, …}`
  - Keys and values need not be the same type!
    - **Example:** Definitions: "set", "1. To put in a specified position..."

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 contains a fixed size array (like vector). It is resized when necessary.

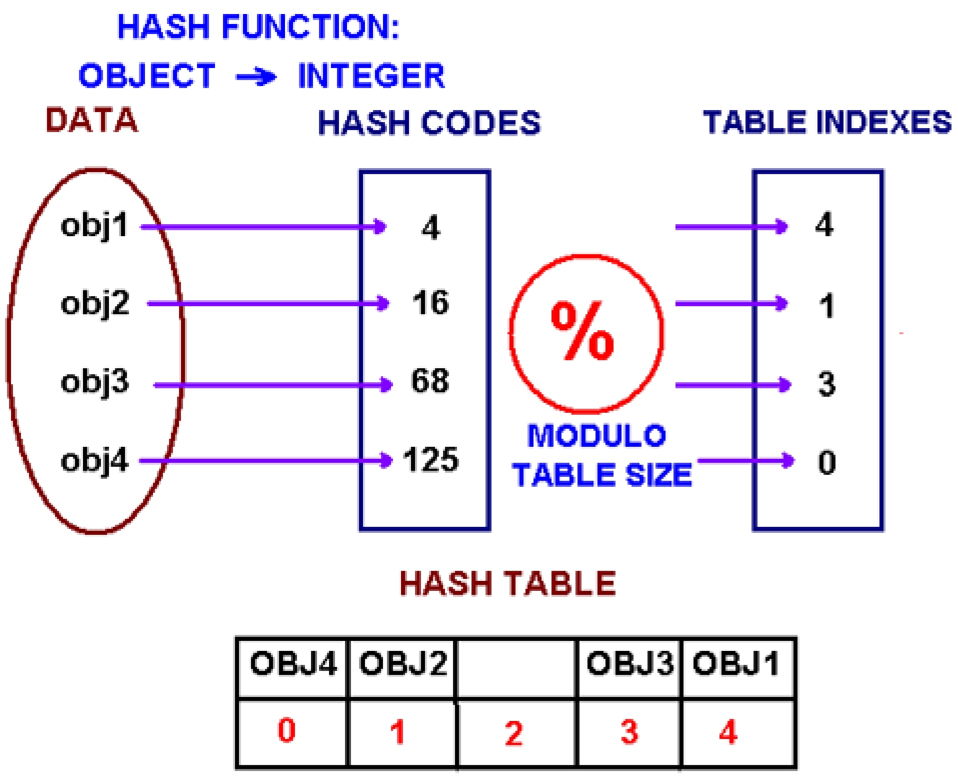
- A hash table uses a **hash function** to compute an **index** into an array of buckets or slots, from which the correct value can be found
  - **Key** passes through a hash function
  - Hash function input: key
  - Hash function output: **index** into the array *(where the value is stored)*

- **A hash table can be searched for an item in O(1) time!** ***(Constant time!)***

What Is A Hash Function?

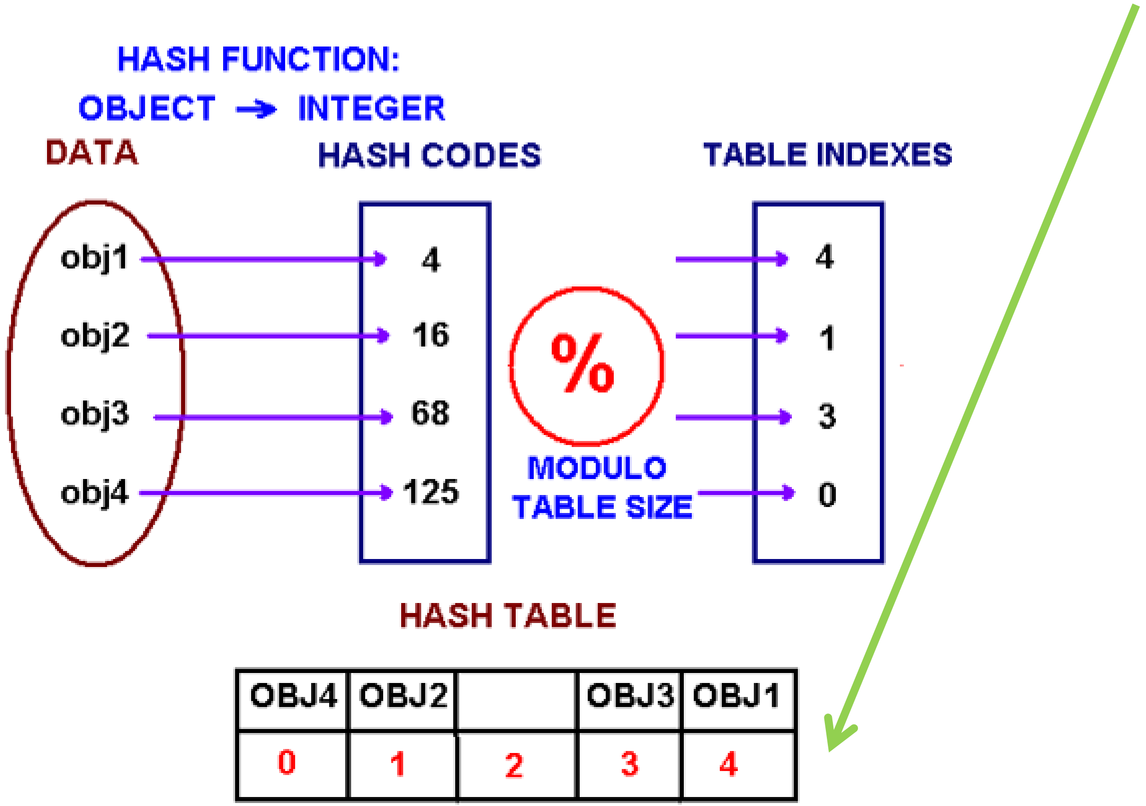
- **Hash function**: a function which, when applied to the **key** (any Java **Object**), produces an [*unsigned* **integer** value **mod** length-of-table] – an integer which can be used as an **address** in a hash table (**index** into an array of *buckets*)

- **Hash functions** have three ***required properties***:
  1. Must be *deterministic* [minimum requirement]
  2. Must be *fast*
  3. Must be *evenly distributed*



What Is A Hash Index?

- **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***



Hash Functions – Raise Your Hand If…

- I'm going to hash all of you into 10 buckets (0-9) by your birthday. (e.g., Nov. 18, 2001)
- The hash functions:
  - By the decade of your birth year
    - *hash*(*birthday*) = (*year*/10) % 10

2001/10 % 10 = 0.1 = 0 (hash index)
1982/10 % 10 = 8.2 = 8

Hash Functions – Raise Your Hand If…

- I'm going to hash all of you into 10 buckets (0-9) by your birthday. (e.g., Nov. 18, 2001)
- The hash functions:
  - By the decade of your birth year
    - *hash*(*birthday*) = (*year*/10) % 10
  - By the last digit of your birth year
    - *hash*(*birthday*) = *year* % 10

2001 % 10 = 1
1982 % 10 = 2

Hash Functions – Raise Your Hand If…

- I'm going to hash all of you into 10 buckets (0-9) by your birthday. (e.g., Nov. 18, 2001)
- The hash functions:
  - By the decade of your birth year
    - *hash*(*birthday*) = (*year*/10) % 10
  - By the last digit of your birth year
    - *hash*(*birthday*) = *year* % 10
  - By the last digit of your birth month
    - *hash*(*birthday*) = *month* % 10

11 % 10 = 1
7 % 10 = 7

Note:
Nov and **Jan**: same hash (1);
Dec and **Feb**: same hash (2).

Hash Functions – Raise Your Hand If…

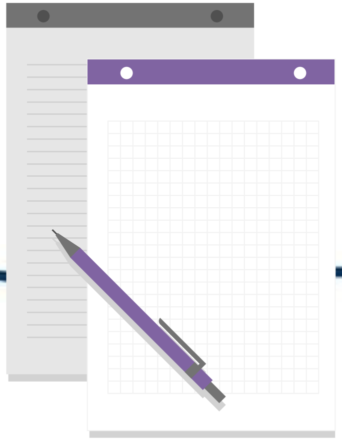
- I'm going to hash all of you into 10 buckets (0-9) by your birthday. (e.g., Nov. 18, 2001)
- The hash functions:
  - By the decade of your birth year
    - *hash*(*birthday*) = (*year*/10) % 10
  - By the last digit of your birth year
    - *hash*(*birthday*) = *year* % 10
  - By the last digit of your birth month
    - *hash*(*birthday*) = *month* % 10
  - By the last digit of your birth day
    - *hash*(*birthday*) = *day* % 10

18 % 10 = 8
23 % 10 = 3

Hash Functions In Java

- Let's look at **Java Object** API
  - https://docs.oracle.com/javase/10/docs/api/java/lang/Object.html
  - Specifically: https://docs.oracle.com/javase/10/docs/api/java/lang/Object.html#hashCode()
- There is a **method** for this!!

Hashing Example (1)



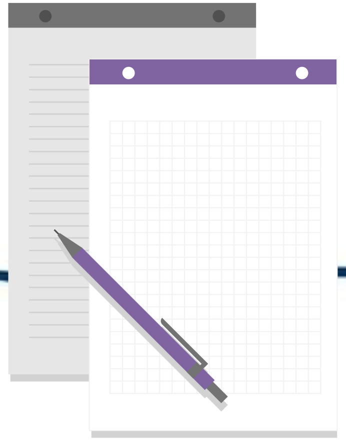
- Key space: integers
- Table size: 10
- `hash(k) = k mod 10`
  - Technically, hash(k) = k, which is then mod'ed by the table size of 10

- **Insert: 7, 18, 41, 34**

0	1	2	3	4	5	6	7	8	9

- How do we find them?

Hashing Example (2)



- Key space: integers
- Table size: 6
- `hash(k) = k mod 6`
  - Size of the hash table is 6 (indices 0 through 5)

- **Insert: 7, 18, 41, 34**

0	1	2	3	4	5

- How do we find them?

Hash Functions

- **Required properties described earlier:**
  1. Must be *deterministic* [minimum requirement]
  2. Must be *fast*
  3. Must be *evenly distributed*

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

- **A "perfect"/ "ideal" hash function:**
  - Will assign each key to a unique bucket (index)
  - No blanks (i.e. no empty cells)
  - No collisions

Rarely achievable in practice!

Hash Function Notes

- They should always return an ***unsigned int***
  - Otherwise, your program will be trying to find a negative array index
- Integer overflow is fine, as long as it overflows *deterministically*
  - Meaning the same way each time (how you handle a 'full' bucket)

- As mentioned, the ideal situation is rarely achievable in practice.
  - 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

Hash Function Notes

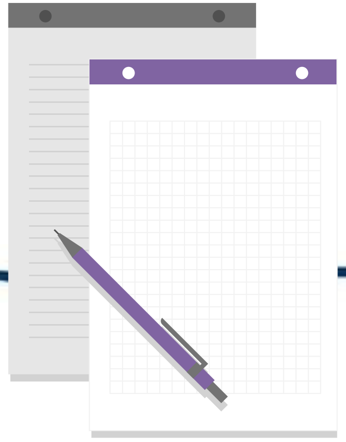
- In a well-dimensioned hash table, the average cost (number of instructions) for each lookup is ***independent*** of the number of elements stored in the table.
- In many situations, hash tables turn out to be more efficient than search trees or any other table lookup structure!
- For this reason, they are widely used in many kinds of computer software, particularly for associative arrays, database indexing, caches, and sets

Collision Resolution

- **Hash collision**: when different keys are hashed (via a hash function) to the same index/ bucket (same location in the hash table)
- Two primary ways to resolve collisions:
  - **Separate Chaining** (make each spot in the table a 'bucket' or a collection)
  - **Open Addressing**, of which there are 3 types:
    - Linear probing
    - Quadratic probing
    - Double hashing

Separate Chaining

Separate Chaining Example (1)



- All keys that map to the same hash value are kept in a "bucket"
  - This "bucket" is another data structure, typically a linked list
- Table size: 10
- `hash(k) = k mod 10`
- **Insert:**
  10, 22, 107, 12, 42
  (0) (2) (7) (2) (2)

0	1	2	3	4	5	6	7	8	9

Analysis of Find

- Definition: The **load factor**, λ, of a hash table is the ratio of the number of elements divided by the table size
- For **separate chaining**, λ is the average number of elements in a bucket
  - Average time on unsuccessful find: λ
    - Average length of a list at hash(k)
  - Average time on successful find: ~ (λ/2)
    - Half the average length of a list (not including the item)

Load Factor

- **How big should we make the hash table?**
- Possible sizes for hash table with separate chaining
  - $\lambda = 1$
    - Make hash table be the number of elements expected; average bucket size is 1
  - $\lambda = 0.75$
    - Good trade-off between memory use and time
  - $\lambda = 0.5$
    - Uses more memory, but fewer collisions

Separate Chaining: Find()

- Given we keep several keys in one bucket when collisions happen, we have to store both the key as well as the value!

- What is the worst case?
  - In the worst case, every key could hash to the same spot!
    - This means it will be a **$\Theta(n)$** algorithm to perform a find!

- What is the "hopeful" case?

What Data Structure To Use For The Buckets?

- **AVL & red-black** trees will give the best running time
  - But that's a lot of overhead!
- **Vectors** are easier, but take up a lot of space
  - All those extra, unused, cells
  - Don't **ever** use vectors for this! ☺
- **Linked lists** are easy, and take up very little space
  - That's **Θ(n)**!
  - Still **faster** *in practice* than **trees** due to having a very small number of items in the bucket

Requirements For The "Hopeful" Case

- **Our ideal hash function and hash table:**
  - Function hash(k) is well distributed for key space
    - For a randomly selected k ∈ K,
    - `probability(hash(k) = i) = 1/table_size`
  - Size of table scales linearly with number of elements
    - Expected bucket size is Θ`(num_elements / table_size)`

- Finding a good hash function can be tough **(Remember ideal hash functions rarely exist!)**
  → A good hash function is very unlikely to return the SAME code for UNEQUAL keys, but MUST return the SAME code for EQUAL keys

  → 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 PERFECT 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.

Separate Chaining Insert Is Θ(1)

- In an **unsorted** linked list, you can just put the newly inserted key on the **front**
- So, all inserts into a separate chained hash table, that uses linked lists, are actually in **constant time**
  - If you insert at the head and you allow duplicates: ***constant time***
  - If you insert at the head and you do **NOT** allow duplicates: ***linear time*** *(to check)*
  - If you were to **sort** the linked list, that would be ***linear time***
  - And **finds** (and thus **deletes**) are still ***linear time***